Separation Logic for Sequential Programs

Arthur Charguéraud

Separation Logic for Sequential Programs (Functional Pearl)

ARTHUR CHARGUÉRAUD, Inria & Université de Strasbourg, CNRS, ICube, France

This paper presents a simple mechanized formalization of Separation Logic for sequential programs. This formalization is aimed for teaching the ideas of Separation Logic, including its soundness proof and its recent enhancements. The formalization serves as support for a course that follows the style of the successful *Software Foundations* series, with all the statement and proofs formalized in Coq. This course only assumes basic knowledge of λ-calculus, semantics and logics, and therefore should be accessible to a broad audience.

CCS Concepts: • **Theory of computation → Separation logic**; *Logic and verification.*

Additional Key Words and Phrases: Separation Logic, Coq, Program verification

ACM Reference Format:
Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 116 (August 2020), 43 pages. https://doi.org/10.1145/3408998

1 INTRODUCTION

Separation Logic brought a major breakthrough in the area of program verification [O'Hearn 2019] (Gödel Prize citation). Since its introduction, Separation Logic has made its way into a number of practical tools that are used on a daily basis for verifying programs ranging from operating systems kernels [Xu et al. 2016] and file systems [Chen et al. 2015] to data structures [Pottier 2017] and graph algorithms [Guéneau et al. 2019]. These programs are written in various programming languages, including machine code [Myreen and Gordon 2007], assembly [Ni and Shao 2006][Chlipala 2013], C-language [Appel and Blazy 2007], OCaml [Charguéraud 2011], SML [Kumar et al. 2014], and Rust [Jung et al. 2017].

The key ideas of Separation Logic were devised by John Reynolds, inspired in part by older work by Burstall [1972]. Reynolds presented his ideas in lectures given in the fall of 1999. The proposed rules turned out to be unsound, but O'Hearn and Ishtiaq [2001] noticed a strong relationship with the logic of *bunched implications* [O'Hearn and Pym 1999], leading to ideas on how to set up a sound program logic. Soon afterwards, the seminal publications on Separation Logic appeared at the CSL workshop [O'Hearn et al. 2001] and at the LICS conference [Reynolds 2002].

Today, when I teach students about Separation Logic, many of them find it hard to believe that Separation Logic has not been around for ever, or at least for as long as program verification exists. Perhaps the best way to truly value Reynold's contribution is to realize that, following the introduction of the first program logics in the late sixties [Floyd 1967; Hoare 1969], people have tried for 30 years to verify programs *without* Separation Logic.

Given its great interest, Separation Logic should presumably be taught to most, if not all, students in the field of programming languages. While Concurrent Separation Logic is a notoriously hard topic [Jung et al. 2018; O'Hearn 2019], Separation Logic for sequential programs is accessible to master students with basic knowledge in logic and semantics. There exists a number of courses

Author's address: Arthur Charguéraud, Inria & Université de Strasbourg, CNRS, ICube, France, arthur.chargueraud@inria.fr.

2475-1421/2020/8-ART116
https://doi.org/10.1145/3408998

that cover Separation Logic for sequential programs, from Reynold's course notes [2006] to modern courses presenting Separation Logic in the context of mechanized proofs, e.g., [Appel 2014; Birkedal and Bizjak 2018; Chlipala 2018a]. However, as we argue in detail in the related work section, existing presentations of Separation Logic either axiomatize the logic and omit important aspects of the soundness proof, or present a soundness proof that involves a number of technical obstacles that get in the way of the students' understanding.

This paper presents a formalization of Separation Logic for sequential programs that, we believe, is simple enough that master students can follow through every detail of its soundness proof, interactively in the Coq proof assistant. Actually, the main contribution of the work described in this paper is a course, entitled *Foundations of Separation Logic* and written in the style of the *Software Foundations* series [Pierce and many contributors 2016]. The present paper consists of a summary of the material from that course, with formal definitions formatted using LaTeX. We refer to the supplementary material [Charguéraud 2020] for the proofs omitted from this paper, as well as numerous additional examples, exercises, comments, and discussion of alternative definitions.

This paper also includes the presentation five key features of Separation Logic that were introduced after the seminal paper from 2002.

(1) The ramified frame rule is a concise and practical rule that combines the frame rule and the rule of consequence into a single rule. The ramified frame rule is essential for automatically simplifying entailments in a robust way (§7.3).
(2) The magic wand operator can be generalized from heap predicates to postconditions. The magic wand operator for postconditions is useful for languages featuring terms with return values, in particular to state the ramified frame rule.
(3) The inductive reasoning rule for loops allows applying the frame rule over the remaining iterations of a loop. This rule typically saves the need to introduce list segments or tree contexts in the statement of loop invariants. In particular, it saves a considerable amount of work when verifying functions that go down a tree-shaped data structure using a while loop. Such functions are pervasive in C implementations of tree-based containers.
(4) The weakest-precondition calculus is, just like in Hoare Logic, a central ingredient for setting up practical verification tools for Separation Logic. In particular, the weakest-precondition style presentation enables better factorization in the statement of reasoning rules, and simplifies the set up of tactics that help instantiating the rules.
(5) The coexistence of *linear* heap predicates and *affine* heap predicates is a must-have for any language equipped with a GC (§8.2). We explain how to set up a Separation Logic with customizable control over which heap predicates are linear and which ones are affine. Such a set up allows to control which heap predicates may be freely discarded.

The features listed above might appear to be somewhat technical, yet there are absolutely essential for taking full advantage of Separation Logic.

The formalization of Separation Logic that we present targets a λ-calculus with imperative features. We consider this language for two reasons. First, the λ-calculus has proved well-suited for teaching programming language semantics in general, as it abstracts away from the details of industrial programming languages. Second, targeting an ML-style language with immutable variables and mutable heap-allocated memory cells leads to the simplest formulation of the reasoning rules, avoiding a number of complications associated with mutable variables. In technical terms, our Coq formalization relies on a standard *deep embedding* of an imperative λ-calculus. This embedding features: an inductive definition of the abstract syntax tree, a recursive definition of the capture-avoiding substitution function, and an inductive definition of the operational semantics. We consider a big-step semantics to simplify the soundness proof.

Our formalization targets the simplest variant of Separation Logic. The heap predicates are defined as higher-order logic predicates, that is, as plain Coq definitions. The reasoning rules of the logic are stated as lemmas that are proved correct with respect to the big-step evaluation rules. In technically terms, our formalization consists of a *shallow embedding* of Separation Logic in Coq. This approach naturally yields a very expressive higher-order Separation Logic. It is employed by the majority of practical verification tools that embed Separation Logic in a proof assistant. It thus makes sense to teach that approach to students that will use or develop those verification tools.

By presenting our course notes on Separation Logic, whose contents are summarized in the present paper, we aim to make the following contributions.

(1) We present a mechanized soundness proof for Separation Logic for sequential programs that we believe to be more accessible to students than previously-available proofs.
(2) We present the first course on Separation Logic written in the style of the *Software Foundations* series. This presentation style, in which every definition and every lemma is presented via its mechanized statement, has been successfully employed for teaching material related to logic and programming languages. Note that the course is focused on the foundations of Separation Logic and the description of its features. It does not (yet) include a large collection of examples illustrating how to specify and verify data structures and algorithms.
(3) We present, within a same paper, five key features that were previously scattered in the literature from the past decade: the ramified frame rule, the magic wand for postconditions, the inductive reasoning technique for loops, the customizable control of affinity, and the weakest-precondition style presentation of the reasoning rules.

This paper is organized as follows. We first give an overview of the key features of Separation Logic (§2). We then present the operators of the logic (§3), the syntax and semantics of the language (§4), the definition of triples and the statement of the reasoning rules (§5). Next, we describe additional techniques: inductive reasoning for loops (§6), the magic wand operator (§7), the generalization to a partially-affine logic (§8), and the presentation in weakest-precondition style (§9). Finally, we give references for all the ideas presented, and discuss related work (§10).

The supplementary material [Charguéraud 2020] accompanying the present submission contains: (1) the full contents of the course, both in Coq and HTML format, (2) a file that contains just the core definitions and soundness proof (SLFMinimal.v), and (3) an appendix to the present paper. This appendix includes statistics on our soundness proof for Separation Logic (§A), background on extensionality axioms (§B), the presentation of the proofs of representative reasoning rules (§C), an example Separation Logic proof (§D), an illustration of the benefits of the frame rule in the proof of recursive functions (§E), a solution to the *cps-append* verification challenge proposed by Reynolds (§F), a discussion of alternative structural rules (§G), a description of the treatment of assertions (§I), of arrays and records (§H), and n-ary functions (§J), and the description of an algorithm for simplifying entailment relations (§K).

2 OVERVIEW OF THE FEATURES OF SEPARATION LOGIC

This section gives an overview of the features that are specific to Separation Logic: the *separating conjunction* and the *frame rule*, which enable *local reasoning* and *small-footprint specifications*, the treatment of aliasing, the specification of recursive pointer-based data structures such as mutable linked lists, and the ability to ensure *complete deallocation* of all allocated data.

2.1 The Frame Rule

In Hoare logic, the behavior of a command t is specified through a *triple*, written $\{H\}\ t\ \{Q\}$, where the *precondition* H describes the input state and the *postcondition* Q describes the output state.

Whereas in Hoare Logic H and Q describe the whole memory state, in Separation Logic they describe only a fragment of the memory state, a fragment that includes all the resources involved in the execution of the command t.

The *frame rule* asserts that if a command t safely executes in a given piece of state, then it also executes safely in a larger piece of state. More precisely, if t executes in a state described by H and produces a final state described by Q, then this program can also be executed in a state that extends H with a *disjoint* piece of state described by H'. The corresponding final state consists of Q extended with H', capturing the fact that the additional piece of state is unmodified by the execution of t. The frame rule enables *local reasoning*, defined as follows [O'Hearn et al. 2001].

> *To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.*

The frame rule is stated using the *separating conjunction*, written $\star$, which is a binary operator over *heap predicates*. In Separation Logic, pieces of states are traditionnally called *heaps*, and predicate over heaps are called *heap predicates*. Given two heap predicates H and H', the heap predicate $H \star H'$ describes a heap made of two disjoint parts, one that satisfies H and one that satisfies H'. The statement of the frame rule, shown below, asserts that any triple remains valid when extending both its precondition and its postcondition with an arbitrary predicate H'.

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \star H'\}}\ \textsc{frame-for-commands} \qquad \text{where } t \text{ is a command.}$$

In this paper, we do not consider a language of commands but a language based on the λ-calculus, with programs described as terms that evaluate to values. (The language is formalized in §4.1.) In that setting, a specification triple takes the form $\{H\}\ t\ \{\lambda x.\ H'\}$, where H describes the input state, x denotes the value produced by the term t, and H' describes the output state, with x bound in H'. For such triples, the frame rule is stated either as:

$$\frac{\{H\}\ t\ \{\lambda x.\ H''\}}{\{H \star H'\}\ t\ \{\lambda x.\ H'' \star H'\}}\ \textsc{frame} \qquad \begin{array}{l}\text{where } t \text{ is a term producing a value,}\\ \text{and } x \notin \mathrm{fv}(H')\end{array}$$

or, more concisely, as:

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \star H'\}}\ \textsc{frame} \qquad \text{where } Q \star H\ \equiv\ \lambda v.\ (Q\,v \star H).$$

2.2 Separation Logic Specifications

What makes Separation Logic work smoothly in practice is that specifications are expressed using a small number of operators for defining heap predicates, such that these operators interact well with the separating conjunctions. The most important operators are summarized below—they appear in examples throughout the rest of this section, and are formally defined further on (§3.2).

- $p \hookrightarrow v$, to be read "p points to v", describes a single memory cell, allocated at address p, with contents v.
- [ ] describes an empty state.
- $[P]$ also describes an empty state, and moreover asserts that the proposition P is true.
- $H_1 \star H_2$ describes a heap made of two disjoint parts, one described by H_1 and another described by H_2.
- $\exists x.\ H$ and $\forall x.\ H$ are used to quantify variables in Separation Logic assertions.

We call these operators the *core heap predicate operators*, because all the other Separation Logic operators can be defined in terms of these core operators.

The heap predicate operators appear in the statement of preconditions and postconditions. For example, consider the specification of the function `incr`, which increments the contents of a reference cell. It is specified using a triple of the form $\{H\}$ (`incr` p) $\{Q\}$.

Example 2.1 (Specification of the increment function).

$$\forall\, p\, n. \qquad \{p \hookrightarrow n\}\ (\texttt{incr}\, p)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}$$

The precondition describes the existence of a memory cell that stores an integer value, through the predicate $p \hookrightarrow n$. The postcondition describes the final heap in the form $p \hookrightarrow (n+1)$, reflecting the increment of the contents. The "$\lambda\_.$ " symbol at the head of the postcondition indicates that the value returned by `incr` p, namely the unit value, needs not be assigned a name.

Throughout the rest of the paper, the outermost universal quantifications (e.g., "$\forall\, p\, n.$") are left implicit, following standard practice.

2.3 Implications of the Frame Rule

The precondition in the specification of `incr` p describes only the reference cell involved in the function call, and nothing else. Consider now the execution of `incr` p in a heap that consists of two distinct memory cells, the first one being described as $p \hookrightarrow n$, and the other being described as $q \hookrightarrow m$. In Separation Logic, the conjunction of these two heap predicates are described by the heap predicate $(p \hookrightarrow n) \star (q \hookrightarrow m)$. There, the separating conjunction (a.k.a. the star) captures the property that the two cells are distinct. The corresponding postcondition of `incr` p describes the updated cell $p \hookrightarrow (n+1)$ as well as the other cell $q \hookrightarrow m$, whose contents is not affected by the call to the increment function. The corresponding Separation Logic triple is therefore stated as follows.

Example 2.2 (Application of the frame rule on the specification of the increment function).

$$\{(p \hookrightarrow n) \star (q \hookrightarrow m)\}\ (\texttt{incr}\, p)\ \{\lambda\_.\ (p \hookrightarrow n+1) \star (q \hookrightarrow m)\}$$

The above triple is derivable from the one stated in Example 2.1 by applying the frame rule to add the heap predicate $q \hookrightarrow m$ both to the precondition and to the postcondition. More generally, any heap predicate H can be added to the original, minimalist specification of `incr` p. Thus we have:

$$\{(p \hookrightarrow n) \star H\}\ (\texttt{incr}\, p)\ \{\lambda\_.\ (p \hookrightarrow n+1) \star H\}.$$

2.4 Treatment of Potentially-Aliased Arguments

We next discuss the case of potentially-aliased reference cells. In the previous example, we have considered two reference cells p and q assumed to be distinct from each other. Consider now a function that expects as arguments two reference cells, at addresses p and q, and increments both. Potentially, the two arguments might correspond to the same reference cell. The function thus admits two specifications. The first one describes the case of two *distinct* arguments, using separating conjunction to assert the difference. The second one describes the case of two *aliased* arguments, that is, the case $p = q$, for which the precondition describes only one reference cell.

Example 2.3 (Potentially aliased arguments). The function:

```
let incr_two p q = (incr p; incr q)
```

admits the following two specifications.

$$\{(p \hookrightarrow n) \star (q \hookrightarrow m)\}\ (\texttt{incr\_two p q})\ \{\lambda\_.\ (p \hookrightarrow n+1) \star (q \hookrightarrow m+1)\}$$
$$\{p \hookrightarrow n\}\ (\texttt{incr\_two p p})\ \{\lambda\_.\ (p \hookrightarrow n+2)\}$$

2.5 Small-Footprint Specifications

A Separation Logic triple captures all the interactions that a term may have with the memory state. Any piece of state that is not described explicitly in the precondition is guaranteed to remain untouched. Separation Logic therefore encourages *small footprint* specifications, i.e., specifications that mention nothing but what is strictly needed. The small-footprint specifications for the primitive operations `ref`, `get` and `set` are stated and explained next.

Example 2.4 (Specification of primitive operations on references).

$$\begin{array}{rll} \{[\,]\} & (\mathsf{ref}\, v) & \{\lambda r.\ \exists p.\ [r = p] \star (p \hookrightarrow v)\} \\ \{p \hookrightarrow v\} & (\mathsf{get}\, p) & \{\lambda r.\ [r = v] \star (p \hookrightarrow v)\} \\ \{p \hookrightarrow v\} & (\mathsf{set}\, p\, v') & \{\lambda \_.\ (p \hookrightarrow v')\} \end{array}$$

The operation `ref` v can execute in the empty state, described by $[\,]$. It returns a value, named r, that corresponds to a pointer p, such that the final heap is described by $p \hookrightarrow v$. In the postcondition, the variable p is quantified existentially, and the pure predicate $[r = p]$ denotes an equality between the value r and the address p, viewed as an element from the grammar of values (formalized in §4.1). The operation `get` p requires in its precondition the existence of a cell described by $p \hookrightarrow v$. Its postcondition asserts that the result value, named r, is equal to the value v, and that the final heap remains described by $p \hookrightarrow v$. The operation `set` $p\, v'$ also requires a heap described by $p \hookrightarrow v$. Its postcondition asserts that the updated heap is described by $p \hookrightarrow v'$. The result value, namely unit, is ignored.

The possibility to state a small-footprint specification for the allocation operation captures an essential property: the reference cell allocated by `ref` is implicitly asserted to be distinct from any pre-existing reference cell. This property can be formally derived by applying the frame rule to the specification triple for `ref`. For example, the triple stated below asserts that if a cell described by $q \hookrightarrow v'$ exists before the allocation operation `ref` v, then the new cell described by $p \hookrightarrow v$ is distinct from that pre-existing cell. This freshness property is captured by the separating conjunction $(p \hookrightarrow v) \star (q \hookrightarrow v')$.

Example 2.5 (Application of the frame rule to the specification of allocation).

$$\{q \hookrightarrow v'\}\ (\mathsf{ref}\, v)\ \{\lambda r.\ \exists p.\ [r = p] \star (p \hookrightarrow v) \star (q \hookrightarrow v')\}$$

The strength of the separating conjunction is even more impressive when involved in the description of recursive data structures such as mutable lists, which we present next.

2.6 Representation of Mutable Lists

A mutable linked list consists of a chain of cells. Each cell contains two fields: the head field stores a value, which corresponds to an item from the list; the tail field stores either a pointer onto the next cell in the list, or the null pointer to indicate the end of the list.

Definition 2.6 (Representation of a list cell). A list cell allocated at address p, storing the value v and the pointer q, is represented by two singleton heap predicates, in the form:

$$(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)$$

where "$p.k$" is a notation for the address $p + k$, and "head ≡ 0" and "tail ≡ 1" denote the offsets.

A mutable linked list is described by a heap predicate of the form Mlist $L\, p$, where p denotes the address of the head cell and L denotes the logical list of the elements stored in the mutable list. The predicate Mlist is called a *representation predicate*, because it relates the pair made of a pointer p and of the heap-allocated data structure that originates at p together with the logical representation of this data structure, namely the list L.

The predicate Mlist is defined recursively on the structure of the list L. If L is the empty list, then p must be null. Otherwise, L is of the form $x :: L'$. In this case, the head field of p stores the item x, and the tail field of p stores a pointer q such that $\mathsf{Mlist}\, L'\, q$ describes the tail of the list. The case disjunction is expressed using Coq's pattern-matching construct.

Definition 2.7 (Representation of a mutable list).

$$\begin{array}{rcl} \mathsf{Mlist}\, L\, p & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ & & \mid x :: L' \;\Rightarrow\; \exists q.\; (p.\mathsf{head} \hookrightarrow x) \;\star\; (p.\mathsf{tail} \hookrightarrow q) \;\star\; (\mathsf{Mlist}\, L'\, q) \end{array}$$

Example 2.8 (Application of the predicate Mlist to a list of length 3). To see how Mlist unfolds on a concrete example, consider the example of a mutable list storing the values 8, 5, and 6.

$$\begin{array}{rcl} \mathsf{Mlist}\, (8 :: 5 :: 6 :: \mathsf{nil})\, p & \equiv & \exists p_1.\; (p.\mathsf{head} \hookrightarrow 8) \;\star\; (p.\mathsf{tail} \hookrightarrow p_1) \\ & & \star\; \exists p_2.\; (p_1.\mathsf{head} \hookrightarrow 5) \;\star\; (p_1.\mathsf{tail} \hookrightarrow p_2) \\ & & \star\; \exists p_3.\; (p_2.\mathsf{head} \hookrightarrow 6) \;\star\; (p_2.\mathsf{tail} \hookrightarrow p_3) \\ & & \star\; [p_3 = \mathsf{null}] \end{array}$$

Observe how the definition of Mlist, by iterating the separating conjunction operator, ensures that all the list cells are distinct from each other. In particular, Mlist precludes the possibility of cycles in the linked list, and precludes inadvertent sharing of list cells with other mutable lists.

Definition 2.7 characterizes Mlist by case analysis on whether the list L is empty. Another, equivalent definition instead characterizes Mlist by case analysis on whether the pointer p is null. This alternative definition is very useful because most list-manipulating programs involve code that tests whether the list pointer at hand is null.

Definition 2.9 (Alternative definition for Mlist).

$$\begin{array}{rcl} \mathsf{Mlist}\, L\, p & \equiv & \mathsf{If}\; (p = \mathsf{null}) \\ & & \quad \mathsf{then}\; [L = \mathsf{nil}] \\ & & \quad \mathsf{else}\; \exists x L' q.\; [L = x :: L'] \;\star\; (p.\mathsf{head} \hookrightarrow x) \;\star\; (p.\mathsf{tail} \hookrightarrow q) \;\star\; (\mathsf{Mlist}\, L'\, q) \end{array}$$

Note that this alternative definition is not recognized as structurally-recursive by Coq. Its statement may be formulated as an equality, and proved correct with respect to Definition 2.7.

2.7 Operations on Mutable Lists

Consider a function that concatenates two mutable lists *in-place*. This function expects two pointers p_1 and p_2 that denote the addresses of two mutable lists described by the logical lists L_1 and L_2, respectively. The first list is assumed to be nonempty. The concatenation operation updates the last cell of the first list so that it points to p_2, the head cell of the second list. After this operation, the mutable list at address p_1 is described by the concatenation $L_1 +\!\!+ L_2$.

Example 2.10 (Specification of in-place append for mutable lists).

$$p_1 \neq \mathsf{null} \;\Rightarrow\; \{(\mathsf{Mlist}\, L_1\, p_1) \;\star\; (\mathsf{Mlist}\, L_2\, p_2)\}\; (\mathtt{mappend}\, p_1\, p_2)\; \{\lambda\_.\;\; \mathsf{Mlist}\, (L_1 +\!\!+ L_2)\, p_1\}$$

Observe how the specification above reflects the fact that the cells of the second list are absorbed by the first list during the operation. These cells are no longer independently available, hence the absence of the representation predicate $\mathsf{Mlist}\, L_2\, p_2$ from the postcondition.

Remark 2.11 (Alternative placement of pure preconditions). The hypothesis $p_1 \neq \mathsf{null}$ from the specification of the append function may be equivalently placed inside the precondition:

$$\{[p_1 \neq \mathsf{null}] \;\star\; (\mathsf{Mlist}\, L_1\, p_1) \;\star\; (\mathsf{Mlist}\, L_2\, p_2)\}\; (\mathtt{mappend}\, p_1\, p_2)\; \{\lambda\_.\;\; \mathsf{Mlist}\, (L_1 +\!\!+ L_2)\, p_1\}.$$

Yet, in general, leaving pure hypotheses as premises outside of triples tends to improve readability.

As second example, consider a function that takes as argument a pointer p to a mutable list, and allocates an entirely independent copy of that list, made of fresh cells. This function is specified as shown below. The precondition describes the input list as $\mathsf{Mlist}\,L\,p$, and the postcondition describes the output heap as $\mathsf{Mlist}\,L\,p \star \mathsf{Mlist}\,L\,p'$, where p' denotes the address of the new list.

Example 2.12 (Specification of a copy function for mutable lists).

$$\{\mathsf{Mlist}\,L\,p\}\ (\mathtt{mcopy}\,p)\ \{\lambda r.\ \exists p'.\ [r = p'] \star (\mathsf{Mlist}\,L\,p) \star (\mathsf{Mlist}\,L\,p')\}$$

The separating conjunction from the postcondition asserts that the original list and its copy do not share any cell: they are entirely disjoint from each other. An implementation and a proof for the function `mcopy` is given in the appendix. The key steps of that proof are summarized next.

Proof. The specification of `mcopy` is proved by induction on the length of the list L. When the list is nonempty, $\mathsf{Mlist}\,L\,p$ unfolds as $(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\,L'\,q)$. The induction hypothesis allows to assume the specification to hold for the recursive call of `mcopy` on the tail of the list, with the precondition $\mathsf{Mlist}\,L'\,q$. Over the scope of that call, the frame rule is used to put aside the head cell, described by $(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)$. Let q' denote the result of the recursive call, and let p' denote the address of a freshly-allocated list cell storing the value x and the tail pointer q'. The final heap is described by:

$$(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\,L'\,q) \star (p'.\mathsf{head} \hookrightarrow x) \star (p'.\mathsf{tail} \hookrightarrow q') \star (\mathsf{Mlist}\,L'\,q')$$

which may be folded to $(\mathsf{Mlist}\,L\,p) \star (\mathsf{Mlist}\,L\,p')$, matching the claimed postcondition. □

In the above proof, the frame rule enables reasoning about a recursive call *independently* of all the cells that have already been traversed by the outer recursive calls to `mcopy`. Without the frame rule, one would have to describe the full list at an arbitrary point during the recursion. Doing so requires describing the *list segment* made of cells ranging from the head of the initial list up to the pointer on which the current recursive call is made. Stating an invariant involving list segments is doable, yet involves more complex definitions and assertions. More generally, for a program manipulating tree-shaped data structures, the frame rule saves the need to describe a tree with a subtree carved out of it, thereby saving a significant amount of proof effort.

2.8 Reasoning about Deallocation

Consider a programming language with explicit deallocation. For such a language, proofs in Separation Logic guarantee two essential properties: (1) a piece of data is never accessed after its deallocation, and (2) every allocated piece of data is eventually deallocated.

The operation $\mathtt{free}\,p$ deallocates the reference cell at address p. This deallocation operation is specified through the following triple, whose precondition describes the cell to be freed by the predicate $p \hookrightarrow v$, and whose postcondition is empty, reflecting the loss of that cell.

Definition 2.13 (Specification of the free operation).

$$\{p \hookrightarrow v\}\ (\mathtt{free}\,p)\ \ \{\lambda\_.\ [\,]\}$$

There is no way to get back the predicate $p \hookrightarrow v$ once it is lost. Because $p \hookrightarrow v$ is required in the precondition of all operations involving the reference p, Separation Logic ensures that no operations on p can be performed after its deallocation.

The next examples show how to specify the deallocation of a list cell and of a full list.

Example 2.14 (Deallocation of a list cell). The function `mfree_cell` deallocates a list cell.

$$\{(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)\}\ (\mathtt{mfree\_cell}\,p)\ \{\lambda\_.\ [\,]\}.$$

Example 2.15 (Deallocation of a mutable list). The function `mfree_list` deallocates a list by recursively deallocating each of its cells. Its implementation is shown below (using ML syntax, even though the language considered features null pointers and explicit deallocation).

```
let rec mfree_list p =
  if p != null then (let q = p.tail in mfree_cell p; mfree_list q)
```

The specification of `mfree_list` admits the precondition $\mathsf{Mlist}\, L\, p$, describing the mutable list to be freed, and admits an empty postcondition, reflecting the loss of that list.

$$\{\mathsf{Mlist}\, L\, p\}\ (\texttt{mfree\_list}\, p)\ \{\lambda\_.\ [\,]\}$$

Remark 2.16 (Languages with implicit garbage collection). For languages equipped with a garbage-collector, Separation Logic can be adapted to allow freely discarding heap predicates (see §8).

3 HEAP PREDICATES AND ENTAILMENT

3.1 Representation of Heaps

Let loc denote the type of locations, i.e., of memory addresses. This type may be realized using, e.g., natural numbers. Let val denote the type of values. The grammar of values depends on the programming language. Its formalization is postponed to §4.

A heap (i.e., a piece of memory state) may be represented as a finite map from locations to values. The finiteness property is required to ensure that fresh locations always exist. Let fmap $\alpha\ \beta$ denote the type of finite maps from a type α to an (inhabited) type β.

Definition 3.1 (Representation of heaps). The type heap is defined as "fmap loc val".

Thereafter, let h denote a heap, that is, a piece of state. Let $h_1 \perp h_2$ assert that two heaps have disjoint domains, i.e., that no location belongs both to the domain of h_1 and to that of h_2. Let $h_1 \uplus h_2$ denote the union of two disjoint heaps. (The union operation may return arbitrary results when applied to non-disjoint arguments, i.e., arguments with overlapping domains.)

3.2 Heap Predicates

A heap predicate, written H, is a predicate that asserts properties of a heap.

Definition 3.2 (Heap predicates). A heap predicate is a predicate of type heap → Prop.

The *core heap predicate operators*, informally introduced in §2.2, are realized as predicates over heaps, as shown below and explained next.

Definition 3.3 (Core heap predicates).

Heap predicate operator	Notation	Definition
empty predicate	$[\,]$	$\lambda h.\ h = \varnothing$
pure fact	$[P]$	$\lambda h.\ h = \varnothing \wedge P$
singleton	$p \mapsto v$	$\lambda h.\ h = (p \to v) \wedge p \neq \mathsf{null}$
separating conjunction	$H_1 \star H_2$	$\lambda h.\ \exists h_1 h_2.\ h_1 \perp h_2 \wedge h = h_1 \uplus h_2 \wedge H_1\, h_1 \wedge H_2\, h_2$
existential quantifier	$\exists x.\, H$	$\lambda h.\ \exists x.\, H\, h$
universal quantifier	$\forall x.\, H$	$\lambda h.\ \forall x.\, H\, h$

The definitions for the core heap predicates all take the form $\lambda h.\ P$, where P denotes a proposition. The empty predicate, written $[\,]$, characterizes a heap equal to the empty heap, written $\varnothing$. The pure predicate, written $[P]$, also characterizes an empty heap, and moreover asserts that the proposition P is true. The singleton heap predicate, written $p \mapsto v$, characterizes a heap described by a singleton map, written $p \to v$, which binds p to v. This predicate embeds the property $p \neq \mathsf{null}$, capturing the

invariant that no data may be allocated at the null location. The separating conjunction, written $H_1 \star H_2$, characterizes a heap h that decomposes as the disjoint union of two heaps h_1 and h_2, with h_1 satisfying H_1 and h_2 satisfying H_2. The existential and universal quantifiers of Separation Logic allow quantifying entities at the level of heap predicates (heap → Prop), in contrast to the standard Coq quantifiers that operate at the level of propositions (Prop). Note that the quantifiers $\exists x.\, H$ and $\forall x.\, H$ may quantify values of any type, without restriction. In particular, they allow quantifying over heap predicates or proof terms.

Remark 3.4 (Encodings between the empty and the pure heap predicate). In Coq, the pure heap predicate $[P]$ can be encoded as "$\exists(p : P).\, [\,]$", that is, by quantifying over the existence of a proof term p for the proposition P. Note that the empty heap predicate $[\,]$ is equivalent to [True].

Remark 3.5 (Other operators). Traditional presentations of Separation Logic include four additional operators, $\bot$, $\top$, $\curlyvee$, and $\curlywedge$. These four operators may be encoded in terms of the ones from Definition 3.3, with the help of Coq's conditional construct. The table below presents the relevant encodings, in addition to providing direct definitions of these operators as predicates over heaps.

Heap predicate operator	Notation	Definition	Encoding
bottom	$\bot$	$\lambda h.$ False	[False]
top	$\top$	$\lambda h.$ True	$\exists(H : \text{heap} \to \text{Prop}).\, H$
disjunction	$H_1 \curlyvee H_2$	$\lambda h.\ (H_1\, h \vee H_2\, h)$	$\exists(b : \text{bool}).$ If b then H_1 else H_2
non-separating conjunction	$H_1 \curlywedge H_2$	$\lambda h.\ (H_1\, h \wedge H_2\, h)$	$\forall(b : \text{bool}).$ If b then H_1 else H_2

Definition 3.6 (Representation predicate for lists defined with disjunction). The representation predicate for lists introduced in Definition 2.8 can be reformulated using the disjunction operator instead of relying on pattern-matching. The corresponding definition, which may be useful if the host logic does not feature a pattern-matching construct, is as follows.

$$\begin{aligned}\text{Mlist}\, L\, p \equiv \quad & \Big([p = \text{null}] \star [L = \text{nil}]\Big) \\ & \curlyvee \Big([p \neq \text{null}] \star \exists x L' q.\ [L = x :: L'] \star (p.\text{head} \hookrightarrow x) \star (p.\text{tail} \hookrightarrow q) \star (\text{Mlist}\, L'\, q)\Big)\end{aligned}$$

3.3 Entailment

The entailment relation, written $H_1 \vdash H_2$, asserts that any heap satisfying H_1 also satisfies H_2.

Definition 3.7 (Entailment relation).

$$H_1 \vdash H_2 \quad \equiv \quad \forall h.\, H_1\, h \Rightarrow H_2\, h$$

Entailment is used to state reasoning rules and to state properties of the heap predicates operators. The entailment relation defines an order relation on the set of heap predicates.

Lemma 3.8 (Entailment defines an order on the set of heap predicates).

$$\textit{HIMPL-REFL}\ \frac{}{H \vdash H} \qquad \textit{HIMPL-TRANS}\ \frac{H_1 \vdash H_2 \qquad H_2 \vdash H_3}{H_1 \vdash H_3} \qquad \textit{HIMPL-ANTISYM}\ \frac{H_1 \vdash H_2 \qquad H_2 \vdash H_1}{H_1 = H_2}$$

The antisymmetry property concludes on an equality between two heap predicates. To establish such an equality, it is necessary to exploit the principle of *predicate extensionality*. This principle asserts that if two predicates P and P', when applied to any argument x, yield logically equivalent propositions, then these two predicates can be considered equal in the logic. (See the appendix for details.) The antisymmetry property plays a critical role for stating the key properties of Separation Logic operators in the form of equalities, as detailed next.

$$\textsc{pure-l}\ \frac{P \Rightarrow (H \vdash H')}{([P] \star H) \vdash H'} \qquad \textsc{exists-l}\ \frac{\forall x.\ (H \vdash H')}{(\exists x.\, H) \vdash H'} \qquad \textsc{forall-l}\ \frac{([a/x]\, H) \vdash H'}{(\forall x.\, H) \vdash H'} \qquad \textsc{exists-monotone}\ \frac{\forall x.\ (H \vdash H')}{(\exists x.\, H) \vdash (\exists x.\, H')}$$

$$\textsc{pure-r}\ \frac{(H \vdash H') \qquad P}{H \vdash (H' \star [P])} \qquad \textsc{exists-r}\ \frac{H \vdash ([a/x]\, H')}{H \vdash (\exists x.\, H')} \qquad \textsc{forall-r}\ \frac{\forall x.\ (H \vdash H')}{H \vdash (\forall x.\, H')} \qquad \textsc{forall-monotone}\ \frac{\forall x.\ (H \vdash H')}{(\forall x.\, H) \vdash (\forall x.\, H')}$$

Fig. 1. Useful properties for pure facts and quantifiers, with respect to entailment.

There are 6 fundamental properties of the separating conjunction operator. The first three capture the fact that $(\star, [\,])$ forms a commutative monoid: the star is associative, commutative, and admits the empty heap predicate as neutral element. The next two describe how quantifiers may be extruded from arguments of the star operator. The extraction rule STAR-EXISTS is stated using an equality because the entailment relation holds in both directions. On the contrary, the extraction rule STAR-FORALL is stated using a simple entailment relation because the reciprocal entailment does not hold—for a counterexample, consider the case where the type of x is unhabited. The rule STAR-MONOTONE-R describes a monotonicity property; it is explained afterwards.

Lemma 3.9 (Fundamental properties of the star).

$$\begin{array}{lrcll}
\textit{STAR-ASSOC:} & (H_1 \star H_2) \star H_3 & = & H_1 \star (H_2 \star H_3) & \\
\textit{STAR-COMM:} & H_1 \star H_2 & = & H_2 \star H_1 & \\
\textit{STAR-NEUTRAL-R:} & H \star [\,] & = & H & \\
\textit{STAR-EXISTS:} & (\exists x.\, H_1) \star H_2 & = & \exists x.\, (H_1 \star H_2) & (\text{if } x \notin H_2) \\
\textit{STAR-FORALL:} & (\forall x.\, H_1) \star H_2 & \vdash & \forall x.\, (H_1 \star H_2) & (\text{if } x \notin H_2)
\end{array}$$

$$\textit{STAR-MONOTONE-R:}\quad \frac{H_1 \vdash H_1'}{H_1 \star H_2 \vdash H_1' \star H_2}$$

The monotonicity rule can be read from bottom to top: when facing a proof obligation of the form $H_1 \star H_2 \vdash H_1' \star H_2$, one may cancel out H_2 on both sides, leaving the proof obligation $H_1 \vdash H_1'$.

Remark 3.10 (Symmetric version of the monotonicity rule). The monotonicity rule is sometimes also presented in its symmetric variant, stated below. It is provably equivalent to STAR-MONOTONE-R.

$$\frac{H_1 \vdash H_1' \qquad H_2 \vdash H_2'}{H_1 \star H_2 \vdash H_1' \star H_2'}\ \textsc{star-monotone}$$

The useful properties associated with pure facts and quantifiers appear in Fig. 1. The application of a number of reasoning rules for entailment can be automated by means of a tactic. (One such tactic is described in the appendix.) Other properties may also be derived, such as $([P_1] \star [P_2]) = [P_1 \wedge P_2]$. Yet, when a simplification tactic is available, one does not need to state such properties explicitly.

The entailment relation may be employed to express how a specific piece of information can be extracted from a given heap predicate. For example, from $p \hookrightarrow v$, one can extract the information $p \neq \text{null}$. Likewise, from a heap predicate of the form $p \hookrightarrow v_1 \star p \hookrightarrow v_2$, where the same location p is described twice, one can derive a contradiction, because the separating conjunction asserts disjointness. These two results are formalized as follows.

Lemma 3.11 (Properties of the singleton heap predicate).

$$\begin{array}{lrcl}
\textit{SINGLE-NOT-NULL:} & (p \hookrightarrow v) & \vdash & (p \hookrightarrow v) \star [p \neq \mathit{null}] \\
\textit{SINGLE-CONFLICT:} & (p \hookrightarrow v_1) \star (p \hookrightarrow v_2) & \vdash & [\mathit{False}]
\end{array}$$

3.4 Generalization to Postconditions

In the imperative λ-calculus considered in this paper and formalized further on (§4), a term evaluates to a value. A postcondition thus describes both an output value and an output state.

Definition 3.12 (Type of postconditions). A postcondition has type: val → heap → Prop.

Thereafter, we let Q range over postconditions. To obtain concise statements of the reasoning rules of Separation Logic for an imperative λ-calculus, it is convenient to extend separating conjunction and entailment to operate on postconditions. To that end, we generalize the predicates $H \star H'$ and $H \vdash H'$ by introducing the predicates $Q \mathbin{\dot\star} H'$ and $Q \mathrel{\dot\vdash} Q'$, written with a dot to suggest *pointwise extension*. These two predicates are formalized next.

Definition 3.13 (Separating conjunction between a postcondition and a heap predicate).

$$Q \mathbin{\dot\star} H \quad \equiv \quad \lambda v.\ (Q\,v \star H)$$

This operator appears for example in the statement of the frame rule (recall §2.1).

The entailment relation for postconditions is a pointwise extension of the entailment relation for heap predicates: Q entails Q' if and only if, for any value v, the heap predicate $Q\,v$ entails $Q'\,v$.

Definition 3.14 (Entailment between postconditions).

$$Q \mathrel{\dot\vdash} Q' \quad \equiv \quad \forall v.\ (Q\,v \vdash Q'\,v)$$

This entailment defines an order on postconditions. It appears for example in the statement of the consequence rule, which allows strengthening the precondition and weakening the postcondition.

Example 3.15 (Rule of consequence).

$$\frac{H \vdash H' \qquad \{H'\}\ t\ \{Q'\} \qquad Q' \mathrel{\dot\vdash} Q}{\{H\}\ t\ \{Q\}}\ \textsc{consequence}$$

4 LANGUAGE SYNTAX AND SEMANTICS

The definition of triples depends on the details of the programming language. Thus, let us first describe the syntax and the semantics of terms.

4.1 Syntax

We consider an imperative call-by-value λ-calculus. The syntactic categories are primitive functions π, values v, and terms t. The grammar of values is intended to denote *closed* values, that is, values without occurrences of free variables. This design choice leads to a simple substitution function, which may be defined as the identity over all values.

The primitive operations fall in two categories. First, they include the state-manipulating operations for allocating, reading, writing, and deallocating references. Second, they include Boolean and arithmetic operations. For brevity, we include only the addition and division operations.

The values include the unit value tt, boolean literals b, integer literals n, memory locations p, primitive operations π, and recursive functions $\hat{\mu} f.\lambda x.t$. The latter construct is written with a hat symbol to denote the fact this value is closed.

The terms include variables, values, function invocation, sequence, let-bindings, conditionals, and function definitions. The latter construct is written $\mu f.\lambda x.t$, this time without a hat symbol.

Definition 4.1 (Syntax of the language).

$$\begin{array}{lcl} \pi & := & \mathsf{ref} \mid \mathsf{get} \mid \mathsf{set} \mid \mathsf{free} \mid (+) \mid (\div) \\ v & := & tt \mid b \mid n \mid p \mid \pi \mid \hat{\mu} f.\lambda x.t \\ t & := & v \mid x \mid (t\,t) \mid \mathsf{let}\, x = t\, \mathsf{in}\, t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mu f.\lambda x.t \end{array}$$

$$\textsc{eval-val}\ \frac{}{v/s \Downarrow v/s} \qquad \textsc{eval-fix}\ \frac{}{(\mu f.\lambda x.t)/s \Downarrow (\hat{\mu} f.\lambda x.t)/s}$$

$$\textsc{eval-app}\ \frac{v_1 = \hat{\mu} f.\lambda x.t \qquad ([v_2/x]\,[v_1/f]\,t)/s \Downarrow v'/s'}{(v_1\, v_2)/s \Downarrow v'/s'}$$

$$\textsc{eval-let}\ \frac{t_1/s \Downarrow v_1/s' \qquad ([v_1/x]\,t_2)/s' \Downarrow v/s''}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)/s \Downarrow v/s''} \qquad \textsc{eval-if}\ \frac{\mathsf{If}\ b\ \mathsf{then}\ (t_1/s \Downarrow v'/s')\ \mathsf{else}\ (t_2/s \Downarrow v'/s')}{(\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow v'/s'}$$

$$\textsc{eval-ref}\ \frac{p \notin \mathrm{dom}\, s}{(\mathsf{ref}\, v)/s \Downarrow p/(s[p := v])} \qquad \textsc{eval-free}\ \frac{p \in \mathrm{dom}\, s}{(\mathsf{free}\, p)/s \Downarrow tt/(s \setminus p)} \qquad \textsc{eval-get}\ \frac{p \in \mathrm{dom}\, s}{(\mathsf{get}\, p)/s \Downarrow (s[p])/s}$$

$$\textsc{eval-set}\ \frac{p \in \mathrm{dom}\, s}{(\mathsf{set}\, p\, v)/s \Downarrow tt/(s[p := v])} \qquad \textsc{eval-add}\ \frac{}{((+)\, n_1\, n_2)/s \Downarrow (n_1 + n_2)/s} \qquad \textsc{eval-div}\ \frac{n_2 \neq 0}{((\div)\, n_1\, n_2)/s \Downarrow (n_1 \div n_2)/s}$$

Fig. 2. Evaluation rules, in big-step style

A non-recursive function $\lambda x.\, t$ may be viewed as a recursive function $\mu f.\lambda x.t$ with a dummy name f. Likewise, a sequence $(t_1\,;\, t_2)$ may be viewed as a let-binding of the form $\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$ for a dummy name x. The Coq formalization actually includes these two constructs explicitly in the grammar to avoid unnecessary complications associated with the elimination of dummy variables.

Although our syntax technically allows for arbitrary terms, for simplicity we assume terms to be written in "administrative normal form" (*A-normal form*), that is, "$\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$" is the sole sequencing construct: no sequencing is implicit in any other construct. For instance, the conditional construct "$\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$" must be encoded as "$\mathsf{let}\, x = t\, \mathsf{in}\, \mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$". This presentation is intended to simplify the statement of the evaluation rules and reasoning rules. Note that many practical program verification tools perform code normalization as a preliminary step.

4.2 Semantics

Thereafter, we use the meta-variable s to denote a variable of type heap that corresponds to a full memory state at a given point in the execution, in contrast to the meta-variable h, which denotes a heap that may correspond to only a piece of the memory state.

The semantics of the language is described by the big-step judgment $t/s \Downarrow v/s'$, which asserts that the term t, starting from the state s, evaluates to the value v and the final state s'.

Definition 4.2 (Semantics of the language). The evaluation rules appear in Fig. 2.

The rules are standard. A value evaluates to itself. Likewise, a function evaluates to itself. The evaluation rules for function calls and let-bindings involve the standard (capture-avoiding) substitution operation: $[v/x]\, t$ denotes the substitution of x by v throughout the term t. The evaluation rule for conditionals is stated concisely using Coq's conditional construct. The primitive operations on reference cells are described using operations on finite maps: $\mathrm{dom}\, s$ denotes the domain of the state s, the operation $s[p]$ returns the value associated with p, the operation $s \setminus p$ removes the binding on p, and the operation $s[p := v]$ sets or updates a binding from p to v.

5 TRIPLES AND REASONING RULES

5.1 Separation Logic Triples

Separation Logic is a refinement of Hoare logic. Interestingly, Separation Logic triples can be defined *in terms of* Hoare triples.

A Hoare triple, written $^{\text{HOARE}}\{H\}\,t\,\{Q\}$, asserts that in any state s satisfying the precondition H, the evaluation of the term t terminates and produces output value v and output state s', as described by the evaluation judgment $t/s \Downarrow v/s'$. Moreover, the output value and output state satisfy the postcondition Q, in the sense that $Q\,v\,s'$ holds. This definition captures termination: it defines a *total correctness* triple. (The definition can be easily adapted to capture *partial correctness* only.)

Definition 5.1 (Total correctness Hoare triple).

$$^{\text{HOARE}}\{H\}\,t\,\{Q\} \quad \equiv \quad \forall s.\, H\,s \;\Rightarrow\; \exists v.\, \exists s'.\, (t/s \Downarrow v/s') \;\wedge\; (Q\,v\,s')$$

Whereas a Hoare triple describes the evaluation of a term with respect to the whole memory state, a Separation Logic triple describes the evaluation of a term with respect to only a fragment of the memory state. To relate the two concepts, it suffices to quantify over "the rest of the state", that is, the part of the state that the evaluation of the term is not concerned with.

A Separation Logic triple, written $\{H\}\ t\ \{Q\}$, asserts that, for any heap predicate H' describing the "rest of the state", the Hoare triple $^{\text{HOARE}}\{H \star H'\}\,t\,\{Q \star H'\}$ holds. This formulation effectively *bakes in* the frame rule, by asserting from the very beginning that specifications are intended to preserve any resource that is not mentioned in the precondition.

Definition 5.2 (Total correctness Separation Logic triple).

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall H'.\ ^{\text{HOARE}}\{H \star H'\}\,t\,\{Q \star H'\}$$

To fully grasp the meaning of a Separation Logic triple, it helps to contemplate an alternative definition expressed directly with respect to the evaluation judgment. This alternative definition, shown below, reads as follows: if the input state decomposes as a part h_1 that satisfies the precondition H and a disjoint part h_2 that describes the rest of the state, then the term t terminates on a value v, producing a heap made of a part h_1' and, disjointly, the part h_2 which was unmodified; moreover, the value v and the heap h_1' together satisfy the postcondition Q.

Definition 5.3 (Alternative definition of total correctness Separation Logic triples).

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall h_1.\, \forall h_2.\, \left\{ \begin{array}{l} H\,h_1 \\ h_1 \perp h_2 \end{array} \right. \;\Rightarrow\; \exists v.\, \exists h_1'.\, \left\{ \begin{array}{l} h_1' \perp h_2 \\ t/(h_1 \uplus h_2) \Downarrow v/(h_1' \uplus h_2) \\ Q\,v\,h_1' \end{array} \right.$$

The reasoning rules of Separation Logic fall in three categories. First, the structural rules: they do not depend on the details of the language. Second, the reasoning rules for terms: there is one such rule for each term construct of the language. Third, the specification of the primitive operations: there is one such rule for each primitive operation. All these rules are presented next.

5.2 Structural Rules

The structural rules of Separation Logic include the consequence rule and the frame rule, which were already discussed, and two rules for extracting pure facts and existential quantifiers out of preconditions. (The role of these rules is illustrated in the example proof presented in the appendix.)

Lemma 5.4 (Structural rules of Separation Logic). *The following reasoning rules can be stated as lemmas and proved correct with respect to the interpretation of triples given by Definition 5.2.*

$$\frac{H \vdash H' \quad \{H'\}\ t\ \{Q'\} \quad Q' \vdash Q}{\{H\}\ t\ \{Q\}}\ \textit{CONSEQUENCE} \qquad \frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \mathbin{\dot\star} H'\}}\ \textit{FRAME} \qquad \frac{P \Rightarrow \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}\ \textit{PROP} \qquad \frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\ H\}\ t\ \{Q\}}\ \textit{EXISTS}$$

The frame rule may be exploited in practice as a *forward* reasoning rule: given a triple $\{H\}\ t\ \{Q\}$, one may derive another triple by extending both the precondition and the postcondition with a heap predicate H'. This rule is, however, almost unusable as a *backward* reasoning rule: indeed, it is extremely rare for a proof obligation to be exactly of the form $\{H \star H'\}\ t\ \{Q \mathbin{\dot\star} H'\}$. In order to exploit the frame rule in backward reasoning, one usually needs to first invoke the consequence rule. The effect of a combined application of the consequence rule followed with the frame rule is captured by the combined *consequence-frame* rule, stated below.

Lemma 5.5 (Combined consequence-frame rule).

$$\frac{H \vdash H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \mathbin{\dot\star} H_2 \vdash Q}{\{H\}\ t\ \{Q\}}\ \textit{CONSEQUENCE-FRAME}$$

This combined rule applies to a proof obligation of the form $\{H\}\ t\ \{Q\}$, with no constraints on the precondition nor the postcondition. To prove this triple from an existing triple $\{H_1\}\ t\ \{Q_1\}$, it suffices to show that the precondition H decomposes as $H_1 \star H_2$, and to show that the postcondition Q can be recovered from $Q_1 \mathbin{\dot\star} H_2$. The "framed" heap predicate H_2 can be computed as the difference between H and H_1. In practice, though, rather than trying to instantiate H_2 in the consequence-frame rule, it is more effective to exploit the ramified frame rule (§7.3).

5.3 Rules for Terms

The program logic includes one rule for each term construct. The corresponding rules are stated below and explained next.

Lemma 5.6 (Reasoning rules for terms in Separation Logic). *The following rules can be stated as lemmas and proved correct with respect to the interpretation of triples given in Definition 5.2.*

$$\frac{H \vdash (Q\ v)}{\{H\}\ v\ \{Q\}}\ \textit{VAL} \qquad \frac{H \vdash (Q\ (\hat\mu f.\lambda x.t))}{\{H\}\ (\mu f.\lambda x.t)\ \{Q\}}\ \textit{FIX} \qquad \frac{v_1 = \hat\mu f.\lambda x.t \qquad \{H\}\ ([v_2/x]\,[v_1/f]\,t)\ \{Q\}}{\{H\}\ (v_1\ v_2)\ \{Q\}}\ \textit{APP}$$

$$\frac{\{H\}\ t_1\ \{\lambda v.\ H'\} \qquad \{H'\}\ t_2\ \{Q\}}{\{H\}\ (t_1\,;\,t_2)\ \{Q\}}\ \textit{SEQ} \qquad \frac{\{H\}\ t_1\ \{Q'\} \qquad \forall v.\ \{Q'\ v\}\ ([v/x]\,t_2)\ \{Q\}}{\{H\}\ (\mathit{let}\, x = t_1\, \mathit{in}\, t_2)\ \{Q\}}\ \textit{LET}$$

$$\frac{b = \mathit{true} \Rightarrow \{H\}\ t_1\ \{Q\} \qquad b = \mathit{false} \Rightarrow \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathit{if}\, b\, \mathit{then}\, t_1\, \mathit{else}\, t_2)\ \{Q\}}\ \textit{IF}$$

The rules VAL and FIX apply to terms that correspond to closed values. A value evaluates to itself, without modifying the state. If the heap at hand is described in the precondition by the heap predicate H, then this heap, together with the value v, should satisfy the postcondition. This implication is captured by the premise $H \vdash Q\,v$. Note that the rules VAL and FIX can also be formulated using triples featuring an empty precondition.

Lemma 5.7 (Small-footprint reasoning rules for values).

$$\frac{}{\{\,[\,]\,\}\ v\ \{\lambda r.\,[r = v]\}}\ \textit{VAL}' \qquad \frac{}{\{\,[\,]\,\}\ (\mu f.\lambda x.t)\ \{\lambda r.\,[r = (\hat\mu f.\lambda x.t)]\}}\ \textit{FIX}'$$

The APP rule merely reformulates the β-reduction rule. It asserts that reasoning about the application of a function to a particular argument amounts to reasoning about the body of this function in which the name of the argument gets substituted with the value of the argument involved in the application. This rule is typically exploited to begin the proof of the specification triple for a function. Once established, such a specification triple may be invoked for reasoning about calls to that function.

The SEQ rule asserts that a sequence "$t_1 \,;\, t_2$" admits precondition H and postcondition Q provided that t_1 admits the precondition H and a postcondition describing a heap satisfying H', and that t_2 admits the precondition H' and the postcondition Q. (The result value v produced by t_1 is ignored.)

The LET rule enables reasoning about a let-binding of the form "let $x = t_1$ in t_2". It reads as follows. Assume that, in the current heap described by H, the evaluation of t_1 produces a postcondition Q'. Assume also that, for any value v that the evaluation of t_1 might produce, the evaluation of $[v/x]\,t_2$ in a heap described by $Q'\,v$ produces the postcondition Q. Then, under the precondition H, the term "let $x = t_1$ in t_2" produces the postcondition Q.

The IF rule enables reasoning about a conditional. Its statement features two premises: one for the case where the condition is the value true, and one for the case where it is the value false.

5.4 Specification of Primitive Operations

The third and last category of reasoning rules corresponds to the specification of the primitive operations of the language. The operations on references have already been discussed (§2.5 and §2.8). The arithmetic operations admit specifications that involve only empty heaps.

LEMMA 5.8 (SPECIFICATION FOR PRIMITIVE OPERATIONS).

$$
\begin{array}{lll}
\textit{REF:} & \{[\,]\}\ (\mathtt{ref}\,v) & \{\lambda r.\ \exists p.\ [r = p] \star (p \hookrightarrow v)\} \\
\textit{GET:} & \{p \hookrightarrow v\}\ (\mathtt{get}\,p) & \{\lambda r.\ [r = v] \star (p \hookrightarrow v)\} \\
\textit{SET:} & \{p \hookrightarrow v\}\ (\mathtt{set}\,p\,v') & \{\lambda \_.\ (p \hookrightarrow v')\} \\
\textit{FREE:} & \{p \hookrightarrow v\}\ (\mathtt{free}\,p) & \{\lambda \_.\ [\,]\} \\
\textit{ADD:} & \{[\,]\}\ ((+)\,n_1\,n_2) & \{\lambda r.\ [r = n_1 + n_2]\} \\
\textit{DIV:} \quad n_2 \neq 0 \ \Rightarrow & \{[\,]\}\ ((\div)\,n_1\,n_2) & \{\lambda r.\ [r = n_1 \div n_2]\}
\end{array}
$$

This completes the presentation of the reasoning rules of Separation Logic. Technically, these 18 reasoning rules suffice to verify imperative programs, although additional infrastructure helps obtain more concise proof scripts. The Coq formalization of the material from Sections §3, §4, and §5 amount to 564 non-blank lines of Coq script. It includes 23 definitions, 59 lemmas, 24 lines of tactic definitions, and 117 lines of proofs. We encourage the reader to check out the corresponding file, which is called SLFMinimal.v in the supplementary material [Charguéraud 2020].

6 INDUCTIVE REASONING FOR LOOPS

Pointer-manipulating programs are typically written using loops. Although loops can be simulated using recursive functions, it simplifies the proofs to include direct reasoning rules for them. Let us assume in this section the presence of a while-loop construct, written "while t_1 do t_2".

A loop "while t_1 do t_2" is equivalent to its one-step unfolding: if t_1 evaluates to true, then t_2 is executed and the loop proceeds; otherwise the loop terminates on the unit value. The rules EVAL-WHILE and WHILE shown below capture this one-step unfolding principle.

LEMMA 6.1 (EVALUATION RULE AND REASONING RULES FOR WHILE LOOPS).

$$
\textit{EVAL-WHILE}\quad \frac{(\mathit{if}\ t_1\ \mathit{then}\ (t_2\,;\ \mathit{while}\,t_1\ \mathit{do}\,t_2)\ \mathit{else}\ \mathit{tt})/s \Downarrow v/s'}{(\mathit{while}\,t_1\ \mathit{do}\,t_2)/s \Downarrow v/s'}
\qquad
\textit{WHILE}\quad \frac{\{H\}\ (\mathit{if}\ t_1\ \mathit{then}\ (t_2\,;\ \mathit{while}\,t_1\ \mathit{do}\,t_2)\ \mathit{else}\ \mathit{tt})\ \{Q\}}{\{H\}\ (\mathit{while}\,t_1\ \mathit{do}\,t_2)\ \{Q\}}
$$

One may establish a triple about the behavior of a while loop by conducting a proof by induction over a decreasing measure or well-founded relation, exploiting the induction hypothesis to reason about the "remaining iterations". Note that this approach is essentially equivalent to encoding the loop as a tail-recursive function, yet without the boilerplate associated with an encoding.

Example 6.2 (Length of a list using a while loop). Consider the following code fragment, which sets the contents of `s` to the length of the mutable list at location `p`.

```
let r = ref p and s = ref 0 in
while !r != null do (incr s; r := !r.tail) done
```

The triple $\{\mathsf{Mlist}\,L\,p \star r \hookrightarrow p \star s \hookrightarrow 0\}$ (`while ... done`) $\{\lambda\_.\, \mathsf{Mlist}\,L\,p \star r \hookrightarrow \mathsf{null} \star s \hookrightarrow |L|\}$ specifies the behavior of the loop. Its proof is conducted by induction on the statement: $\forall L n p.\ \{\mathsf{Mlist}\,L\,p \star r \hookrightarrow p \star s \hookrightarrow n\}$ (`while ... done`) $\{\lambda\_.\, \mathsf{Mlist}\,L\,p \star r \hookrightarrow \mathsf{null} \star s \hookrightarrow n + |L|\}$. Applying the WHILE rule reveals the conditional on whether `!r` is null. In the case where it is not null, `s` is incremented, `r` is set to the tail of the current list, and the loop starts over. To reason about this "recursive invocation" of the while-loop, it suffices to apply the frame rule to put aside the head cell described by a predicate of the form $(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)$, and to apply the induction hypothesis to the tail of the list described by $\mathsf{Mlist}\,L'\,q$, where $L = x :: L'$.

The above example shows that, by carrying a proof by induction, it is possible to apply the frame rule over the remaining iterations of a loop. Doing so would not be possible with a reasoning rule that imposes a loop invariant to be valid both at the entry point and exit point of the loop body. Indeed, such a loop invariant would necessarily involve the description of a list segment.

7 THE MAGIC WAND OPERATOR

7.1 Definition and Properties of the Magic Wand

The magic wand, also known as *separating implication*, is an additional heap predicate operator, written $H_1 \mathrel{-\!\!\star} H_2$, and read "$H_1$ wand H_2". Although it is technically possible to carry out all Separation Logic proofs without the magic wand, this operator helps to state several reasoning rules and specifications more concisely.

Intuitively, $H_1 \mathrel{-\!\!\star} H_2$ defines a heap predicate such that, if starred with H_1, it produces H_2. In other words, the magic wand satisfies the *cancellation rule* $H_1 \star (H_1 \mathrel{-\!\!\star} H_2) \vdash H_2$. The magic wand operator can be formally defined in at least four different ways.

Definition 7.1 (Magic wand). The magic wand operator is equivalently characterized by:

(1) $H_1 \mathrel{-\!\!\star} H_2 \;\equiv\; \lambda h.\ \big(\forall h'.\ h \perp h' \wedge H_1\,h' \Rightarrow H_2\,(h \uplus h')\big)$

(2) $H_1 \mathrel{-\!\!\star} H_2 \;\equiv\; \exists H_0.\ H_0 \star \big[(H_1 \star H_0) \vdash H_2\big]$

(3) $H_0 \vdash (H_1 \mathrel{-\!\!\star} H_2) \;\Leftrightarrow\; (H_1 \star H_0) \vdash H_2$

(4) $H_1 \mathrel{-\!\!\star} H_2$ satisfies the following introduction and elimination rules.

$$\frac{(H_1 \star H_0) \vdash H_2}{H_0 \vdash (H_1 \mathrel{-\!\!\star} H_2)}\ \textsc{wand-intro} \qquad \frac{}{H_1 \star (H_1 \mathrel{-\!\!\star} H_2) \vdash H_2}\ \textsc{wand-cancel}$$

The first characterization asserts that $H_1 \mathrel{-\!\!\star} H_2$ holds of a heap h if and only if, for any disjoint heap h' satisfying H_1, the union of the two heaps $h \uplus h'$ satisfies H_2.

The second characterization describes a heap satisfying a predicate H_0 that, when starred with H_1 entails H_2. This characterization shows that the magic wand can be encoded using previously-introduced concepts from higher-order Separation Logic.

The third characterization consists of an equivalence that provides both an introduction rule and an elimination rule. The left-to-right direction is equivalent to the cancellation rule WAND-CANCEL stated in definition (4). The right-to-left direction corresponds exactly to the introduction rule from definition (4), namely WAND-INTRO, which reads as follows: to show that a heap described by H_0 satisfies the magic wand $H_1 \star H_2$, it suffices to prove that H_1 starred with H_0 entails H_2.

In practice, the following properties are useful for working with the magic wand and for implementing a tactic that simplifies the proof obligations that arise from the *ramified frame rule* (§7.3).

LEMMA 7.2 (USEFUL PROPERTIES OF THE MAGIC WAND).

WAND-MONOTONE
$$\frac{H_1' \vdash H_1 \qquad H_2 \vdash H_2'}{(H_1 \mathrel{-\!\!\star} H_2) \vdash (H_1' \mathrel{-\!\!\star} H_2')}$$

WAND-SELF
$$\frac{}{[\,] \vdash (H \mathrel{-\!\!\star} H)}$$

WAND-PURE-L
$$\frac{P}{([P] \mathrel{-\!\!\star} H) = H}$$

WAND-CURRY
$$\frac{}{((H_1 \star H_2) \mathrel{-\!\!\star} H_3) = (H_1 \mathrel{-\!\!\star} (H_2 \mathrel{-\!\!\star} H_3))}$$

WAND-STAR
$$\frac{}{((H_1 \mathrel{-\!\!\star} H_2) \star H_3) \vdash (H_1 \mathrel{-\!\!\star} (H_2 \star H_3))}$$

LEMMA 7.3 (PARTIAL CONCELLATION OF A MAGIC WAND). *If the left-hand side of a magic wand involves the separating conjunction of several heap predicates, it is possible to cancel out just one of them with an occurrence of the same heap predicate occurring outside of the magic wand. For example, the entailment* $H_2 \star \big((H_1 \star H_2 \star H_3) \mathrel{-\!\!\star} H_4\big) \vdash \big((H_1 \star H_3) \mathrel{-\!\!\star} H_4\big)$ *is obtained by cancelling* H_2.

7.2 Magic Wand for Postconditions

Just as useful as the magic wand is its generalization to postconditions, which is involved for example in the statement of the ramified frame rule (§7.3). This operator, written $Q_1 \mathrel{-\!\!\star} Q_2$, takes as argument two postconditions Q_1 and Q_2 and produces a heap predicate.

Definition 7.4 (Magic wand for postconditions). The operator $(\mathrel{-\!\!\star})$ is equivalently defined by:

(1) $Q_1 \mathrel{-\!\!\star} Q_2 \;\equiv\; \forall v.\ \big((Q_1\, v) \mathrel{-\!\!\star} (Q_2\, v)\big)$

(2) $Q_1 \mathrel{-\!\!\star} Q_2 \;\equiv\; \lambda h.\ \big(\forall v h'.\ h \perp h' \wedge Q_1\, v\, h' \Rightarrow Q_2\, v\, (h \uplus h')\big)$

(3) $Q_1 \mathrel{-\!\!\star} Q_2 \;\equiv\; \exists H_0.\ H_0 \star \big[(Q_1 \mathbin{\dot\star} H_0) \vdash Q_2\big]$

(4) $H_0 \vdash (Q_1 \mathrel{-\!\!\star} Q_2) \;\Leftrightarrow\; (Q_1 \mathbin{\dot\star} H_0) \vdash Q_2$

(5) $Q_1 \mathrel{-\!\!\star} Q_2$ satisfies the following introduction and elimination rules.

$$\frac{(Q_1 \mathbin{\dot\star} H_0) \vdash Q_2}{H_0 \vdash (Q_1 \mathrel{-\!\!\star} Q_2)}\ \text{QWAND-INTRO} \qquad \frac{}{Q_1 \mathbin{\dot\star} (Q_1 \mathrel{-\!\!\star} Q_2) \vdash Q_2}\ \text{QWAND-CANCEL}$$

LEMMA 7.5 (USEFUL PROPERTIES OF THE MAGIC WAND FOR POSTCONDITIONS).

QWAND-MONOTONE
$$\frac{Q_1' \vdash Q_1 \qquad Q_2 \vdash Q_2'}{(Q_1 \mathrel{-\!\!\star} Q_2) \vdash (Q_1' \mathrel{-\!\!\star} Q_2')}$$

QWAND-SELF
$$\frac{}{[\,] \vdash (Q \mathrel{-\!\!\star} Q)}$$

QWAND-STAR
$$\frac{}{((Q_1 \mathrel{-\!\!\star} Q_2) \mathbin{\dot\star} H) \vdash (Q_1 \mathrel{-\!\!\star} (Q_2 \mathbin{\dot\star} H))}$$

QWAND-SPECIALIZE
$$\frac{}{\big(Q_1 \mathrel{-\!\!\star} Q_2\big) \vdash \big((Q_1\, v) \mathrel{-\!\!\star} (Q_2\, v)\big)}$$

7.3 Ramified Frame Rule

One key practical application of the magic wand operator appears in the statement of the *ramified frame rule*. This rule reformulates the consequence-frame rule in a manner that is both more

concise and better-suited for automated processing. Recall the rule CONSEQUENCE-FRAME, which is reproduced below. To exploit it, one must provide a predicate H_2 describing the "framed" part. Providing the heap predicate H_2 by hand in proofs involves a prohibitive amount of work; it is strongly desirable that H_2 may be inferred automatically.

The predicate H_2 can be computed as the difference between H and H_1. Automatically computing this difference is relatively straightforward in simple cases, however this task becomes quite challenging when H and H_1 involve numerous quantifiers. Indeed, it is not obvious to determine which quantifiers from H should be cancelled against those from H_1, and which quantifiers should be carried over to H_2.

The benefit of the ramified frame rule is that it eliminates the problem altogether. The key idea is to observe that the premise $Q_1 \mathbin{\bigstar} H_2 \vdash Q$ from the CONSEQUENCE-FRAME rule is equivalent to $H_2 \vdash (Q_1 \mathrel{-\!\!\bigstar} Q)$, by the 4th characterization of Definition 7.4. Thus, in the other premise $H \vdash H_1 \bigstar H_2$, the heap predicate H_2 may be replaced with $Q_1 \mathrel{-\!\!\bigstar} Q$. The RAMIFIED-FRAME rule appears below.

LEMMA 7.6 (RAMIFIED FRAME RULE). *RAMIFIED-FRAME reformulates CONSEQUENCE-FRAME.*

CONSEQUENCE-FRAME

$$\frac{H \vdash H_1 \bigstar H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \mathbin{\bigstar} H_2 \vdash Q}{\{H\}\ t\ \{Q\}}$$

RAMIFIED-FRAME

$$\frac{\{H_1\}\ t\ \{Q_1\} \qquad H \vdash H_1 \bigstar (Q_1 \mathrel{-\!\!\bigstar} Q)}{\{H\}\ t\ \{Q\}}$$

8 PARTIALLY-AFFINE SEPARATION LOGIC

8.1 Linear and Affine Heap Predicates

The Separation Logic presented so far is well-suited for a language with explicit deallocation. It is, however, impractical for a language equipped with a garbage collector. Indeed, it does not provide any rule for discarding the description of pieces of state that are ready for the garbage collector to dispose of. The aim of this section is to refine the definitions presented so far to support rules that enable discarding certain heap predicates from either the precondition or the postcondition.

The formalization presented is general enough to allow fine-tuning between *affine* heap predicates, which may be freely discarded, and *linear* heap predicates, which, on the contrary, must remain accounted for. For example, linearity is useful to ensure that every file handle opened eventually gets closed, or to ensure that every lock acquired eventually gets released. To that end, the reasoning rules should not allow discarding the heap predicates that represent linear resources.

In technical terms, a Separation Logic is said to be *linear* if no heap predicates can be discarded—like in original presentations of Separation Logic. A Separation Logic is said to be *affine* if any heap predicates may be freely discarded at any time. The purpose of this section is to set up a *partially-affine* Separation Logic, in which both *linear* and *affine* heap predicates may coexist.

8.2 Customizable Characterization of Affine Heap Predicates

The predicate affine H asserts that the heap predicate H may be freely discarded. This predicate is defined in terms of a lower-level predicate, written haffine h, that characterizes which heaps may be discarded. This predicate is axiomatized. Two specific instantiations are presented: one that treats all heaps as discardable, leading to a *fully-affine* logic, and one that treats none of them as discardable, leading to a *fully-linear* logic, equivalent to the logic developed so far.

Definition 8.1 (Axiomatization of affine heaps). The predicate haffine h must satisfy two rules:

$$\frac{}{\mathsf{haffine}\ \varnothing}\ \textsc{staffine-empty} \qquad \frac{\mathsf{haffine}\ h_1 \qquad \mathsf{haffine}\ h_2 \qquad h_1 \perp h_2}{\mathsf{haffine}\ (h_1 \uplus h_2)}\ \textsc{staffine-union}$$

The predicate affine H characterizes heap predicates that hold only of affine heaps.

Definition 8.2 (Definition of affine heap predicates).

$$\mathsf{affine}\,H \quad \equiv \quad \forall h.\ H\,h \ \Rightarrow\ \mathsf{haffine}\,h$$

The rules presented next establish that the composition of affine heap predicates yield affine heap predicates. In other words, the predicate affine is stable by composition. For example, a heap predicate $H_1 \star H_2$ is affine provided that H_1 and H_2 are both affine. A heap predicate $\exists x.\ H$ is affine provided that H is affine for any variable x. Likewise, a heap predicate $\forall x.\ H$ is affine provided that H is affine for any variable x, with a technical restriction asserting that the type of x must be inhabited (because, otherwise, the hypothesis would be vacuous).

Lemma 8.3 (Sufficient conditions for affinity of a heap predicate).

$$\frac{}{\mathit{affine}\,[\,]}\ \textit{AFFINE-EMPTY} \qquad \frac{}{\mathit{affine}\,[P]}\ \textit{AFFINE-PURE} \qquad \frac{\mathit{affine}\,H_1 \qquad \mathit{affine}\,H_2}{\mathit{affine}\,(H_1 \star H_2)}\ \textit{AFFINE-STAR}$$

$$\frac{\forall x.\ \mathit{affine}\,H}{\mathit{affine}\,(\exists x.\ H)}\ \textit{AFFINE-EXISTS} \qquad \frac{\forall x.\ \mathit{affine}\,H \qquad \text{the type of } x \text{ is inhabited}}{\mathit{affine}\,(\forall x.\ H)}\ \textit{AFFINE-FORALL} \qquad \frac{P \ \Rightarrow\ \mathit{affine}\,H}{\mathit{affine}\,([P] \star H)}\ \textit{AFFINE-STAR-PURE}$$

In practice, the application of these rules is automated using a tactic, thus the process of justifying that a heap predicate is affine is in most cases totally transparent for the user.

To state the reasoning rules that enable discarding affine heap predicates, it is helpful to introduce the *affine top* heap predicate, which is written $\mathbb{T}$. Whereas the top heap predicate (written $\top$ and defined as "$\lambda h.\ \mathsf{true}$") holds of *any heap*, the affine top predicate holds only of *any affine heap*.

Definition 8.4 (Affine top). The predicate $\mathbb{T}$ can be equivalently defined in two ways.

$$(1) \quad \mathbb{T} \ \equiv\ \lambda h.\ \mathsf{haffine}\,h \qquad\qquad (2) \quad \mathbb{T} \ \equiv\ \exists H.\ [\mathsf{affine}\,H] \star H$$

There are three important properties of $\mathbb{T}$. The first one asserts that any affine heap predicate H entails $\mathbb{T}$. The second one asserts that the predicate $\mathbb{T}$ is itself affine. The third one asserts that several copies of $\mathbb{T}$ are equivalent to a single $\mathbb{T}$.

Lemma 8.5 (Properties of affine top).

$$\frac{\mathit{affine}\,H}{H \vdash \mathbb{T}}\ \textit{ATOP-R} \qquad \frac{}{\mathit{affine}\,\mathbb{T}}\ \textit{AFFINE-ATOP} \qquad \frac{}{(\mathbb{T} \star \mathbb{T}) \ =\ \mathbb{T}}\ \textit{STAR-ATOP-ATOP}$$

All the aforementioned definitions and lemmas hold for any predicate haffine satisfying the axiomatization from Definition 8.1. Two extreme instantiations of haffine are particularly interesting.

Example 8.6 (Fully-affine Separation Logic). The definition "$\mathsf{haffine}\,h \equiv \mathsf{True}$" satisfies the requirements of Definition 8.1, and leads to a Separation Logic where all heap predicates may be freely discarded. In that setting, $(\mathsf{affine}\,H) \ \Leftrightarrow\ \mathsf{True}$, and $\mathbb{T} = \top = (\lambda h.\ \mathsf{true}) = (\exists H.\ H)$.

Example 8.7 (Fully-linear Separation Logic). The definition "$\mathsf{haffine}\,h \equiv (h = \varnothing)$" satisfies the requirements of Definition 8.1, and leads to a Separation Logic where no heap predicate may be freely discarded. In that setting, $(\mathsf{affine}\,H) \ \Leftrightarrow\ (H \vdash [\,])$, and $\mathbb{T} = [\,] = (\lambda h.\ h = \varnothing)$.

8.3 Triples for a Partially-Affine Separation Logic

To accommodate reasoning rules that enable freely discarding affine heap predicates, it suffices to refine the definition of a Separation Logic triple (Definition 5.2) by integrating the affine top predicate $\mathbb{T}$ into the postcondition of the underlying Hoare triple, as formalized next.

Definition 8.8 (Refined definition of triples for Separation Logic).

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall H'.\ ^{\text{HOARE}}\{H \star H'\}\ t\ \{Q \mathbin{\dot\star} H' \mathbin{\dot\star} \top\}$$

Note that, with the fully-linear instantiation described in Example 8.7, the predicate $\top$ is equivalent to the empty heap predicate, therefore Definition 8.8 is strictly more general than Definition 5.2.

Lemma 8.9 (Reasoning rules for refined Separation Logic triples). *All the previously-mentioned reasoning rules, in particular the structural rules (Lemma 5.4) and the reasoning rules for terms (Lemma 5.6), remain correct with respect to the refined definition of triples (Definition 8.8).*

The *discard rules*, which enable discarding affine heap predicates, may be stated in a number of ways. The three variants that are most useful in practice are shown below. These three variants have equivalent expressive power with respect to discarding heap predicates.

The rule DISCARD-PRE allows discarding a user-specified predicate H' from the precondition, provided that H' is affine. Without this rule, the user would have to carry this heap predicate H' through the proof until it appears in a postcondition.

The rule ATOP-POST allows extending the postcondition with $\top$, allowing a subsequent proof step to yield an entailment relation of the form $Q_1 \vdash (Q \mathbin{\dot\star} \top)$, allowing to discard unwanted pieces from Q_1. This rule is useful in "manual" proofs, i.e., proofs carried out with limited tactic support.

The rule RAMIFIED-FRAME-ATOP extends the ramified frame rule so that its entailment integrates the predicate $\top$, allowing to discard unwanted pieces from either H or Q_1. This rule is a key building block for a practical tool that implements a partially-affine Separation Logic.

Lemma 8.10 (Discard rules).

$$\textit{DISCARD-PRE}\ \frac{\{H\}\ t\ \{Q\} \qquad \mathit{affine}\, H'}{\{H \star H'\}\ t\ \{Q\}} \qquad \textit{ATOP-POST}\ \frac{\{H\}\ t\ \{Q \mathbin{\dot\star} \top\}}{\{H\}\ t\ \{Q\}} \qquad \textit{RAMIFIED-FRAME-ATOP}\ \frac{\{H_1\}\ t\ \{Q_1\} \qquad H \vdash H_1 \star (Q_1 \mathbin{\dot{-\!\star}} (Q \mathbin{\dot\star} \top))}{\{H\}\ t\ \{Q\}}$$

9 WEAKEST-PRECONDITION STYLE

9.1 Semantic Weakest Precondition

The notion of weakest precondition has been used pervasively in the development of practical tools based on Hoare logic. Recent work has shown that this notion also helps streamlining the set up of practical tools based on Separation Logic.

The *semantic weakest precondition* of a term t with respect to a postcondition Q denotes a heap predicate, written $\mathrm{wp}\, t\, Q$, which corresponds to the *weakest* precondition H satisfying the triple $\{H\}\ t\ \{Q\}$. The notion of "weakest" is to be understood with respect to the entailment relation, which induces an order relation on the set of heap predicates (recall Lemma 3.8). The definition of the predicate wp can be formalized in at least five different ways. The corresponding definitions are shown below and commented next.

Definition 9.1 (Semantic weakest precondition). The predicate wp is equivalently characterized by:

(1) $\quad \mathrm{wp}\, t\, Q \ \equiv\ \min_{(\vdash)} \left\{\ H \ \middle|\ \{H\}\ t\ \{Q\}\ \right\}$

(2) $\quad \Big(\{\mathrm{wp}\, t\, Q\}\ t\ \{Q\}\Big) \ \wedge\ \Big(\forall H.\ \{H\}\ t\ \{Q\} \ \Rightarrow\ H \vdash \mathrm{wp}\, t\, Q\Big)$

(3) $\quad \mathrm{wp}\, t\, Q \ \equiv\ \lambda h.\ \Big(\{\lambda h'.\, h' = h\}\ t\ \{Q\}\Big)$

(4) $\quad \mathrm{wp}\, t\, Q \ \equiv\ \exists H.\ H \star \Big[\{H\}\ t\ \{Q\}\Big]$

(5) $\quad H \vdash \mathrm{wp}\, t\, Q \ \Leftrightarrow\ \{H\}\ t\ \{Q\}$

The first characterization asserts that $\mathrm{wp}\, t\, Q$ is *the weakest precondition*: it is a valid precondition for a triple for the term t with the postcondition Q. Moreover, any other valid precondition H for a

triple involving t and Q entails $\mathsf{wp}\, t\, Q$. The second characterization consists of a reformulation of the first characterization in terms of basic logic operators.

The third characterization defines $\mathsf{wp}\, t\, Q$ as a predicate over a heap h, asserting that $\mathsf{wp}\, t\, Q$ holds of the heap h if and only if the evaluation of the term starting from a heap *equal to* h produces the postcondition Q.

The fourth characterization asserts that $\mathsf{wp}\, t\, Q$ is entailed by any heap predicate H satisfying the triple $\{H\}\ t\ \{Q\}$. This characterization shows that the notion of weakest precondition can be expressed as a derived notion in terms of the core heap predicate operators.

The fifth characterization asserts that any triple of the form $\{H\}\ t\ \{Q\}$ may be equivalently reformulated by replacing this triple with $H \vdash \mathsf{wp}\, t\, Q$.

The developer of a practical tool based on Separation Logic may choose to take either triples or weakest-preconditions as a primitive notion; the other notion may then be derived in terms of that primitive notion. The notion of triple is typically defined in terms of wp using characterization (5), in the right-to-left direction. Reciprocally, the definition of wp can be defined in terms of triples. The choice of the encoding depends on the strength of the host logic with respect to existential quantification. Definition (3) makes weaker assumptions, whereas Definition (4) leverages the ability to existentially quantify over heap predicates. Definition (4), which is expressed at the level of heap predicates, is generally much simpler to manipulate in proofs.

9.2 WP-Style Structural Rules

The structural reasoning rule can be reformulated in weakest-precondition style, as follows.

Lemma 9.2 (Structural rules in weakest precondition style).

$$\frac{Q \vdash Q'}{wp\, t\, Q \vdash wp\, t\, Q'}\ \textit{WP-CONSEQUENCE} \qquad \frac{}{(wp\, t\, Q) \star H \vdash wp\, t\, (Q \star H)}\ \textit{WP-FRAME}$$

$$\frac{\textit{affine}\, H}{(wp\, t\, Q) \star H \vdash (wp\, t\, Q)}\ \textit{WP-DISCARD-PRE} \qquad \frac{}{wp\, t\, (Q \star \top) \vdash wp\, t\, Q}\ \textit{WP-ATOP-POST}$$

The rule WP-CONSEQUENCE captures a monotonicity property. The rule WP-FRAME reads as follows: *if I own a heap in which the execution of t produces the postcondition Q, and, separately, I own a heap satisfying H, then, altogether, I own a heap in which the execution of t produces both Q and H.* These four structural rules may be combined into a single rule, called WP-RAMIFIED-FRAME-ATOP, which subsumes all the other structural rules of Separation Logic.

Lemma 9.3 (Ramified frame rule in weakest precondition style).

$$\frac{}{(wp\, t\, Q) \ \star\ (Q \mathrel{-\!\!\star}\ (Q' \star \top)) \vdash (wp\, t\, Q')}\ \textit{WP-RAMIFIED-FRAME-ATOP}$$

9.3 WP-Style Rules For Terms

The weakest-precondition style reformulation of the reasoning rules for terms yields rules that are similar to the corresponding Hoare logic rules. For example, the rule for sequence is as follows.

$$\frac{}{\mathsf{wp}\, t_1\, (\lambda v.\ \mathsf{wp}\, t_2\, Q) \ \vdash\ \mathsf{wp}\, (t_1\, ;\, t_2)\, Q}\ \textsc{wp-seq}$$

This rule can be read as follows: *if I own a heap in which the execution of t_1 produces a heap in which the execution of t_2 produces the postcondition Q, then I own a heap in which the execution of the sequence "$t_1\, ;\, t_2$" produces Q.* The other reasoning rules for terms appear below.

Lemma 9.4 (Reasoning rules for terms in weakest precondition style).

$$\frac{}{Q\,v \;\vdash\; wp\,v\,Q}\;\textit{WP-VAL} \qquad \frac{}{Q\,(\hat{\mu}f.\lambda x.t) \;\vdash\; wp\,(\mu f.\lambda x.t)\,Q}\;\textit{WP-FIX}$$

$$\frac{v_1 = \hat{\mu}f.\lambda x.t}{wp\,([v_2/x]\,[v_1/f]\,t)\,Q \;\vdash\; wp\,(v_1\,v_2)\,Q}\;\textit{WP-APP}$$

$$\frac{}{wp\,t_1\,(\lambda v.\; wp\,([v/x]\,t_2)\,Q) \;\vdash\; wp\,(\mathit{let}\,x = t_1\,\mathit{in}\,t_2)\,Q}\;\textit{WP-LET}$$

$$\frac{}{\mathit{If}\,b\;\mathit{then}\;(wp\,t_1\,Q)\;\mathit{else}\;(wp\,t_2\,Q) \;\vdash\; wp\,(\mathit{if}\,b\;\mathit{then}\;t_1\;\mathit{else}\;t_2)\,Q}\;\textit{WP-IF}$$

9.4 WP-Style Function Specifications

Function specifications were so far expressed using triples of the form $\{H\}\ (f\,v)\ \{Q\}$. These specifications may be equivalently expressed using assertions of the form $H \vdash \mathrm{wp}\,(f\,v)\,Q$.

The primitive operations are specified using wp as shown below. For example, the allocation operation $\mathrm{ref}\,v$ produces a postcondition Q, provided that the result of extending the current precondition with $p \hookrightarrow v$ yields $Q\,p$. In the formal statement of the specification WP-REF, observe how the address p is quantified universally in the left-hand side of the entailment.

Lemma 9.5 (Specification of primitive operations in weakest-precondition style).

$$\begin{array}{lll}
\textit{WP-REF}: & \forall\, Q\,v. & \big(\forall p.\ (p \hookrightarrow v) \mathrel{-\!\!\star} (Q\,p)\big) \vdash wp\,(\mathrm{ref}\,v)\,Q \\
\textit{WP-GET}: & \forall\, Q\,p. & (p \hookrightarrow v) \star \big((p \hookrightarrow v) \mathrel{-\!\!\star} (Q\,v)\big) \vdash wp\,(\mathrm{get}\,p)\,Q \\
\textit{WP-SET}: & \forall\, Q\,p\,v\,v'. & (p \hookrightarrow v) \star \big(\forall r.\ (p \hookrightarrow v') \mathrel{-\!\!\star} (Q\,r)\big) \vdash wp\,(\mathrm{set}\,p\,v')\,Q \\
\textit{WP-FREE}: & \forall\, Q\,p\,v. & (p \hookrightarrow v) \star \big(\forall r.\ (Q\,r)\big) \vdash wp\,(\mathrm{free}\,p)\,Q
\end{array}$$

Remark: WP-SET and WP-FREE can also be stated by specializing the variable r to the unit value tt.

There exists a general pattern for translating from conventional triples to weakest-precondition style specifications. The following lemma covers the case of a specification involving a single auxiliary variable named x. It may easily be generalized to a larger number of auxiliary variables.

Lemma 9.6 (Specifications in weakest-precondition style). *Let v denote a value that may depend on a variable x, and let H' denote a heap predicate that may depend on the variables x and r.*

$$\big(\{H\}\ t\ \{\lambda r.\ \exists x.\ [r = v] \star H'\}\big) \;\Leftrightarrow\; \big(\forall Q.\ H \star (\forall x.\ H' \mathrel{-\!\!\star} (Q\,v)) \;\vdash\; wp\,t\,Q\big)$$

Stating specifications in weakest-precondition style is not at all mandatory for working with reasoning rules in weakest-precondition style. Indeed, one may continue stating specifications using conventional triples, which one might find more intuitive to read, and exploit the following rule for reasoning about function applications.

Lemma 9.7 (Variant of the ramified frame rule for proof obligations in wp style).

$$\frac{\{H_1\}\ t\ \{Q_1\} \qquad H \;\vdash\; H_1 \star (Q_1 \mathrel{-\!\!\star} Q)}{H \;\vdash\; wp\,t\,Q}\;\textit{RAMIFIED-FRAME-FOR-WP}$$

10 RELATED WORK

For a broad survey of Separation Logic, we refer to O'Hearn's CACM paper [2019]. In particular, its appendix covers practical automated and semi-automated tools based on Separation Logic, such as Infer [Calcagno et al. 2015], VeriFast [Philippaerts et al. 2014], or Viper [Müller et al. 2016]. In this related work section, we focus on tools that leverage Separation Logic in interactive proofs assistants for verifying sequential programs, and on the comparison with other teaching material.

10.1 Original Presentation of Separation Logic

Traditional presentations of Separation Logic target command-based languages, which involve mutable variables in addition to heap-allocated data. In that setting, the statement of the frame rule involves a side-condition to assert that the mutable variables occurring in the framed heap predicate are not modified by the command. Up to minor differences in presentation, many fundamental concepts appeared in the first descriptions of Separation Logic [O'Hearn et al. 2001; Reynolds 2002]:

- the grammar of heap predicate operators, except the pure heap predicate [P], and with the limitation that quantifiers $\exists x.\, H$ and $\forall x.\, H$ range only over integer values;
- the rule of consequence and the frame rules;
- a variant of the rule EXISTS (this variant is named EXISTS2 in the appendix);
- the fundamental properties of the star operator described in Lemma 3.9;
- the small footprint specifications for primitive state-manipulating operations,
- the definition of Mlist, stated by pattern-matching over the list structure like in Definition 2.7;
- the characterization of the magic wand operator via characterizations (1), (3) and (4) from Definition 7.1, but not characterization (2), which involves quantification over heap predicates;
- the example of a copy function for binary trees;
- the encoding of records and arrays using pointer arithmetics (described in the appendix).

10.2 Additional Features of Separation Logic

The original presentation of Separation Logic consists of a first-order logic for a first-order language.

Biering et al. [2005, 2007] tackled the generalization to *higher-order quantification*—the possibility to quantify over propositions and heap predicates—through the introduction of *BI-hyperdoctrines*. Krishnaswami et al. [2007] formalized the subject-observer pattern with a strong form of information hiding between the subject and the client. This work illustrated how higher-order Separation Logic supports data abstraction.

Birkedal et al. [2005, 2006] tackled the generalization of Separation Logic to *higher-order languages*, where functions may take functions as arguments. To avoid complications with mutable variables, the authors considered a version of Algol with immutable variables and first-order heaps—heap cells can only store integer values. Specifications are presented using dependent types: a triple $\{H\}\ t\ \{Q\}$ is expressed by the fact that the term t admits the type "$\{H\} \cdot \{Q\}$". One key idea from this work is to bake-in the frame rule into the interpretation of triples, that is, to quantify over a heap predicate describing the rest of the state, as in Definition 5.2. The technique of the baked-in frame rule later proved successful in mechanized proofs. For example, it appears in the HOL4 formalization by Myreen and Gordon [2007] (see §3.2, as well as §2.4 from Myreen's PhD thesis [2008]) and in the Coq formalization by [Appel and Blazy 2007] (see Definition 9).

Reus and Schwinghammer [2006] presented a generalization of Separation Logic to *higher-order stores*, where heap cells may store functions whose execution may act over the heap. The former work targets a language that features storable, parameter-less procedures. Its model, developed on paper, was then simplified by Birkedal et al. [2008] using the technique of the baked-in frame rule.

Another approach to tackling the circularity issues associated with higher-order quantification and higher-order stores consists of using the *step indexing* technique [Ahmed 2004; Appel and McAllester 2001; Appel et al. 2007]. In that approach, a heap predicate depends not only on a heap but also on a natural number, which denotes the number of execution steps for which the predicate is guaranteed to hold. This approach was later exploited in VST, which provided the first higher-order *concurrent* Separation Logic [Hobor et al. 2008].

Ni and Shao [2006] presented the XCAP framework, formalized in Coq. It targets an assembly-level language with embedded code pointers, thereby supporting both higher-order functions

and higher-order stores. XCAP features *impredicative polymorphism*, allowing heap predicates to quantify over heap predicates. This work addresses the same problem as the aforementioned work through a more syntactic approach.

When reasoning about first-class functions, the notion of *nested triple* naturally appears: triples may occur inside the pre- or post-condition of other triples. Nested triples were described in work by [Schwinghammer et al. 2009] for functions stored in the heap, and in work by [Svendsen et al. 2010] for higher-order functions (more precisely, for delegate functions). Nested triples are naturally supported by shallow embeddings of Separation Logic in higher-order logic proof assistants. This possibility is mentioned explicitly by [Wang et al. 2011], but was already implicitly available in earlier formalizations, e.g. [Appel and Blazy 2007].

Krishnaswami et al. [2010] introduced the idea of a ramified frame rule. The general statement of the ramified rule stated as in Lemma 7.6 appeared in Hobor and Villard [2013]. Users of the tools VST [Cao et al. 2018b] and Iris [Jung et al. 2017] have advertised for the interest of this rule.

The magic wand between postconditions, written $Q_1 \mathrel{-\!\!\star} Q_2$, as opposed to the use of an explicit quantification $\forall v.\, Q_1\, v \mathrel{-\!\!\star} Q_2\, v$, appears to have first been employed by Bengtson et al. [2012]. This operator is described in the book by Appel et al. [2014]. The five equivalent characterizations of this operator give in Definition 7.4 appear to be a (very minor) contribution of the present paper.

Regarding while loops, the possibility to frame over the remaining iterations (§6) is inherently available when a loop is encoded as a recursive functions, or when a loop is presented in CPS-style—typical with assembly-level code [Chlipala 2011; Ni and Shao 2006]. The statement of a reasoning rule directly applicable to a non-encoded loop construct, and allowing to frame over the remaining iterations, has appeared independently in work by Charguéraud [2010] and Tuerk [2010].

A number of interesting extensions of Separation Logic for deterministic sequential programs were beyond the scope of the present paper. Let us cite a few.

The notion of *Separation Algebra* [Calcagno et al. 2007; Dockins et al. 2009; Gotsman et al. 2011; Klein et al. 2012] is useful for developing a Separation Logic framework independently from the details of the programming language. Costanzo and Shao [2012] present a refined definition of *local reasoning* to ensure that, whenever a program runs safely on some state, adding more state would have no effect on the program's behavior; their definition is useful in particular for nondeterministic programs and programs executed in a finite memory. *Fictional Separation Logic* [Jensen and Birkedal 2012] generalizes the interpretation of separating conjunction beyond physical separation, and explains how to combine several separation algebras. *Temporary read-only permissions* [Charguéraud and Pottier 2017] provide a simpler alternative to fractional permission for manipulating duplicatable read-only resources in a sequential program. *Time credits* [Charguéraud and Pottier 2015] allow for amortized cost analysis. *Time receipts* [Mével et al. 2019] may be used to establish lower bounds on the execution time. The *higher-order frame* [Birkedal et al. 2005, 2006] and the *higher-order anti-frame* [Pottier 2008; Schwinghammer et al. 2010] allow reasoning about hidden state in sequential programs.

10.3 Mechanized Presentations of Separation Logic

Gordon [1989] presents the first mechanization of Hoare logic in higher-order logic, using the HOL tool. Gordon's pioneering work was followed by numerous formalizations of Hoare logic, targeting various programming languages. Mechanizations of Separation Logic appeared later. Here again, we restrict our discussion to the verification of sequential programs.

Yu et al. [2003, 2004] present the CAP framework, implemented in Coq. It supports reasoning about low-level code using Separation Logic-style rules, and is applied to the verification of a dynamic storage allocation library. Ni and Shao [2006] present the XCAP framework, already mentioned in the previous section, to reason about embedded code pointers. XCAP was also applied

to reasoning about x86 context management code [Ni et al. 2007]. Feng et al. [2006] present the SCAP framework, for reasoning about stack-based control abstractions, including exceptions and setjmp/longjmp operations. SCAP is also applied to the verification of Baker's incremental copying garbage collector [McCreight et al. 2007]. Feng et al. [2007] present the OCAP framework that generalizes XCAP for supporting interoperability of different verification systems, including SCAP. Cai et al. [2007] present the GCAP framework for reasoning about self-modifying code, and apply Separation Logic to support local reasoning on both program code and regular data structures. Feng et al. [2008] presents the first verified implementation of a preemptive thread runtime that exploits hardware interrupts; this runtime is linked to verified context switch primitives, using the OCAP and the SCAP frameworks. Wang et al. [2011] present ISCAP, a step-indexed, direct-style operational semantics with support for first-class pointers.

Weber [2004] formalizes in Isabelle/HOL a first-order Separation Logic for a simple while language. This work includes a soundness proof for the frame rule, and the verification of the classic in-place list reversal example.

Preoteasa [2006] formalize in PVS a first-order Separation Logic, with the additional feature that it supports recursive procedures. This work includes the verification of a collection of recursive procedures for computing the parse tree associated with an arithmetic expression.

Marti et al. [2006] formalize in Coq a Separation Logic library, and used it for the verification of the heap manager of an operating system.

Tuch et al. [2007] present a shallow embedding of Separation Logic in Isabelle/HOL, for a subset of the C language, with support for interpreting values at the byte level when required. Their framework is applied to the verification of the memory allocator of a microkernel. Its logic was later extended to support predicates for mapping virtual to physical addresses, and thereby reason about the effects of virtual memory [Kolanski and Klein 2009]. Klein et al. [2012] present a re-usable library for Separation Algebras, including support for automation.

Appel and Blazy [2007] formalize in Coq a Separation logic for Cminor. This work led to the VST tool, which supports the verification of concurrent C code [Appel 2011; Appel et al. 2014; Cao et al. 2018a]. VST leverages step-indexed definitions and features a *later modality* [Dockins et al. 2008; Hobor et al. 2008, 2010].

Myreen and Gordon [2007] formalize Separation Logic in HOL4. This work eventually lead to the CakeML compiler, described further on.

Varming and Birkedal [2008] demonstrate the possibility to formalize *higher-order* Separation Logic as a shallow embedding in Isabelle/HOLCF.

Nanevski et al. [2008b] and Chlipala et al. [2009] present the Ynot tool, which consists of an axiomatic embedding in Coq of Hoare Type Theory (HTT) [Nanevski et al. 2006, 2008a]. HTT is a presentation of higher-order Separation Logic with higher-order stores in the form of a type system for a dependently typed functional language. In Ynot, like in HTT, a Coq term t admits the Coq type "ST H Q" to express the specification $\{H\}\ t\ \{Q\}$. In Ynot, programs are shallowly embedded in Coq: they are expressed using Coq primitive constructs and axiomatized monadic constructs for effects. The frame rule takes the form of an identity coercion of type $\mathsf{ST}\,H\,Q \to \mathsf{ST}\,(H \star H')\,(\lambda v.\ Q\,v \star H')$. For specifications involving auxiliary variables, Ynot supports *ghost* arguments, which appear like normal function arguments except that they are erased at runtime.

Charguéraud [2011] presents the CFML tool, which supports the verification of OCaml programs. CFML does not state reasoning rules directly in Coq; instead, a program is verified by means of its *characteristic formula*, which corresponds to a form of strongest postcondition. These characteristic formulae are generated as Coq axioms by an external tool that parses input programs in OCaml syntax. CFML was extended to support asymptotic cost analysis [Charguéraud and Pottier 2015;

Charguéraud and Pottier 2019]. CFML initially hard-wired fully-affine triples, featuring unrestricted discard rules, and later integrated the customizable predicate haffine (§8) [Guéneau et al. 2019].

Tuerk [2011] presents in HOL4 the *Holfoot* tool, formalizing in particular the rules of *Abstract Separation Logic* [Calcagno et al. 2007].

Chlipala [2011, 2013] presents in Coq the Bedrock framework, for the verification of programs written at the assembly level. Bedrock has been, for example, put to practice to verify a cooperative threading library and an implementation of a domain-specific language for XML processing. These software components were interfaced with hardware components of mobile robots [Chlipala 2015].

Bengtson et al. [2011] present a shallow embedding of higher-order Separation Logic in Coq, demonstrating the use of nested triples for reasoning about object-oriented code. Following up on that work, Bengtson et al. [2012] developed in Coq the *Charge!* tool, which handles a subset of Java.

Jensen et al. [2013] give a modern presentation of a Separation Logic for low-level code, exploiting in particular the (higher-order) frame connective [Birkedal et al. 2005; Birkedal and Yang 2007; Krishnaswami 2012]. Building on that work, Kennedy et al. [2013] show how to write assembly syntax and generate x86 machine code inside Coq.

The CakeML verified compiler [Kumar et al. 2014], implemented in HOL, takes SML-like programs as input and produces machine code as output. It exploits Separation Logic to prove the garbage collector [Sandberg Ericsson et al. 2019]. It also exploits Separation Logic to set up a CFML-style characteristic formula generator, extended with support for catchable exceptions and I/O [Guéneau et al. 2017]. The characteristic formulae are used to verify the standard library for CakeML.

The Iris framework [Jung et al. 2017, 2016, 2018, 2015; Krebbers et al. 2017], implemented in Coq, supports higher-order concurrent Separation Logic. Like VST, Iris features a later modality and step-indexed definitions. Iris exploits weakest-precondition style reasoning rules (§9) and function specifications are stated as in Lemma 9.6, although using syntactic sugar to make specifications resemble conventional triples. Iris is defined as a fully-affine logic, with an affine entailment. Tassarotti et al. [2017] present an extension of Iris with linear heap predicates.

The Mosel framework [Krebbers et al. 2018] generalizes Iris' tooling to a large class of separation logics, targeting both affine and linear separation logics, and combinations thereof. On top of Iris, Bizjak et al. [2019] present the encoding of two logics that enable tracking of linear resources that are transferable among dynamically allocated threads. The first one, called Iron, leverages fractional permissions to encode *trackable resources*, and allow, e.g., reasoning about deallocation of shared resources. The second one, called Iron++, hides away the use of fractions, and recovers essentially a *linear* Separation Logic with support for *trackable invariants*.

Bannister et al. [2018] discuss techniques for forward and backward reasoning in Separation Logic. Their work, presented in Isabelle/HOL, introduces the *separating coimplication* operator to improve automation. Separating coimplication is the dual of separating conjunction, just like *septraction* [Vafeiadis and Parkinson 2007] is the dual of separating implication. Separating coimplication forms a Galois connection with septraction, just like separating conjunction forms a Galois connection with separating implication.

Lammich [2019a,b] present a refinement framework that leverages Separation Logic to refine from Isabelle/HOL definitions to verified code in LLVM intermediate representation. It is applied to the production of a number of algorithms, including an efficient KMP string search implementation.

10.4 Tutorials on Separation Logic

There exists a number of course notes on Separation Logic. Many of them follow the presentation from Reynolds' article [2002] and course notes [2006]. These course notes consider languages with mutable variables, whose treatment adds complexity to the reasoning rules. The Separation Logic is presented as a first-order logic on its own, without attempt to relate it in a way or another to the

higher-order logic of a proof assistant. The soundness of the logic is generally only skimmed over, with a few lines explaining how to justify the frame rule.

A few courses present Separation Logic in relation with its application in mechanized proofs. Appel's book *Program Logics For Certified Compilers* [2014] presents a formalization of a Separation Logic targeting the C semantics from CompCert [Leroy 2009]. More recently, Appel and Cao [2020] published a volume part of the Software Foundations series, entitled *Verifiable C*. This volume is a tutorial for VST [Cao et al. 2018a], a tool that supports reasoning about actual C code. As of 2020, the tutorial covers the verification of data structures, including linked lists, stacks, hashtables, as well as functions manipulating strings.

The presence of mutable variables, in addition to other specificities of the C memory model, makes the presentation unnecessarily complex for a first exposure to Separation Logic and to its soundness proof. The author is aware of two other mechanized Separation Logic tutorials that target a λ-calculus based language, with immutable variables and return values for terms.

The Iris tutorial by Birkedal and Bizjak [2018] presents the core ideas of Iris' concurrent Separation Logic [Krebbers et al. 2017]. Chapters 3 and 4 introduce heap predicates and Separation Logic for sequential programs. Unlike in Iris' Coq formalization, which leverages a shallow embedding of Separation Logic, the tutorial presents the heap predicate in deep embedding style, via a set of typing rules for heap predicates. The realization of these predicates is not explained, and the tutorial does not discuss how the reasoning rules are proved sound with respect to the small-step semantics of the language. The logic presented targets partial correctness, not total correctness, and only the case of an affine logic is covered.

Chlipala's course notes [Chlipala 2018a] feature a chapter on Separation Logic, accompanied with a corresponding Coq formalization meant to be followed by students [Chlipala 2018b]. The material includes a proof of soundness, as well as the verification of a few example programs. Chlipala's chapter focuses on the core of Separation Logic—it does not cover any of the enhancements listed in the introduction. The programming language is described in *mixed-embedding* style: the syntax includes a constructor Bind, which represents bindings using Coq functions, in higher-order abstract syntax style. The rest of the syntax consists of operations for allocation and deallocation, for reading and writing integer values into the heap, plus the constructors Return, Loop, and Fail. These constructs are dependently-typed: a term that produces a value of type α admits the type cmd α. Altogether, this design allows for a concise formalization of the source language, yet, we believe, at the price of an increased cost of entry for the reader unfamiliar with the techniques involved. The core heap predicates are formalized like in Ynot [Chlipala et al. 2009]. Triples are defined in deep embedding style, via an inductive definition whose constructors correspond to the reasoning rules. This deep embedding presentation requires "not-entirely-obvious" inversion lemmas, which are not needed in our approach. The soundness proof establishes a partial correctness result expressed via preservation and progress lemmas. Chlipala's approach appears well suited for reasoning about an operating system kernel that should never terminate, or reasoning about concurrent code. However, for reasoning about sequential executions of functions that do terminate, our total correctness proof carried out with respect to a big-step semantics yields a stronger result, via a simpler proof.

ACKNOWLEDGMENTS

I wish to thank François Pottier and Armaël Guéneau, with whom I have worked on extensions of Separation Logic. I am grateful to Jacques-Henri Jourdan, who explained to me a number of techniques used in Iris. I also wish to thank the anonymous reviewers for their useful suggestions, as well as Andrew Appel, Lars Birkedal, Adam Chlipala, Magnus Myreen, Gerwin Klein, Peter Lammich, and Zhong Shao, who helped complete the related work section. This research was partly supported by the French National Research Organization (project VOCAL ANR-15-CE25-008).

REFERENCES

Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-28891-3_2

Andrew W Appel. 2014. *Program logics for certified compilers.* Cambridge University Press. https://doi.org/10.1017/CBO9781107256552 With Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy.

Andrew W Appel and Sandrine Blazy. 2007. Separation logic for small-step Cminor. In *International Conference on Theorem Proving in Higher Order Logics*, Klaus Schneider and Jens Brandt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–21. https://doi.org/10.1007/978-3-540-74591-4_3

Andrew W. Appel and Qinxiang Cao. 2020. *Verifiable C.* Software Foundations, Vol. 5beta. Electronic textbook. http://softwarefoundations.cis.upenn.edu Version 0.9.5.

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press, USA. https://doi.org/10.1017/CBO9781107256552

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/1190216.1190235

Callum Bannister, Peter Höfner, and Gerwin Klein. 2018. Backwards and Forwards with Separation Logic. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 68–87. https://doi.org/10.1007/978-3-319-94821-8_5

Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. 2012. Charge!. In *Interactive Theorem Proving*, Lennart Beringer and Amy Felty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 315–331. https://doi.org/10.1007/978-3-642-32347-8_21

Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. 2011. Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq. In *Interactive Theorem Proving*, Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 22–38. https://doi.org/10.1007/978-3-642-22863-6_5

Bodil Biering, Lars Birkedal, and Noah Torp-Smith. 2005. BI Hyperdoctrines and Higher-Order Separation Logic. In *Proceedings of the 14th European Conference on Programming Languages and Systems* (Edinburgh, UK) *(ESOP'05)*. Springer-Verlag, Berlin, Heidelberg, 233–247. https://doi.org/10.1007/978-3-540-31987-0_17

Bodil Biering, Lars Birkedal, and Noah Torp-Smith. 2007. BI-Hyperdoctrines, Higher-Order Separation Logic, and Abstraction. *ACM Trans. Program. Lang. Syst.* 29, 5 (Aug. 2007), 24–es. https://doi.org/10.1145/1275497.1275499

Lars Birkedal and Aleš Bizjak. 2018. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. https://iris-project.org/tutorial-material.html

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. 2008. A Simple Model of Separation Logic for Higher-Order Store. In *Automata, Languages and Programming (ICALP)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–360. https://doi.org/10.1007/978-3-540-70583-3_29

Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. 2005. Semantics of separation-logic typing and higher-order frame rules. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE, 260–269. https://doi.org/10.1109/LICS.2005.47

Lars Birkedal, Noah Torp-smith, and Hongseok Yang. 2006. Semantics of separation-logic typing and higher-order frame rules for algol-like languages, PrakashEditor Panangaden (Ed.). *Logical Methods in Computer Science* 2, 5. https://doi.org/10.2168/lmcs-2(5:1)2006

Lars Birkedal and Hongseok Yang. 2007. Relational Parametricity and Separation Logic. In *Foundations of Software Science and Computational Structures*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 93–107. https://doi.org/10.1007/978-3-540-71389-0_8

Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing Obligations in Higher-Order Concurrent Separation Logic. *Proc. ACM Program. Lang.* 3, POPL, Article 65 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290378

R. M. Burstall. 1972. Some Techniques for Proving Correctness of Programs which Alter Data Structures. In *Machine Intelligence 7*, B. Meltzer and D. Mitchie (Eds.). Edinburgh University Press, Edinburgh, Scotland., 23–50.

Hongxu Cai, Zhong Shao, and Alexander Vaynberg. 2007. Certified Self-Modifying Code. *SIGPLAN Not.* 42, 6 (June 2007), 66–77. https://doi.org/10.1145/1273442.1250743

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *Logic in Computer Science (LICS)*. 366–378. https://doi.org/10.1109/LICS.2007.30

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. 2018a. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W. Appel. 2018b. Proof pearl: Magic wand as frame. Unpublished.

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 418–430. https://doi.org/10.1145/2034773.2034828

Arthur Charguéraud and François Pottier. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 137–153. https://doi.org/10.1007/978-3-319-22102-1_9

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning (JAR)* 62, 3 (March 2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

Arthur Charguéraud. 2010. *Characteristic Formulae for Mechanized Program Verification.* Ph.D. Dissertation. Université Paris Diderot. http://www.chargueraud.org/research/2010/thesis/thesis_final.pdf

Arthur Charguéraud. 2020. Supplementary material. http://www.chargueraud.org/teach/verif/

Arthur Charguéraud and François Pottier. 2017. Temporary Read-Only Permissions for Separation Logic. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 260–286. https://doi.org/10.1007/978-3-662-54434-1_10

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

Adam Chlipala. 2011. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. *SIGPLAN Not.* 46, 6 (June 2011), 234–245. https://doi.org/10.1145/1993316.1993526

Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier, In Proceedings of the 18th ACM SIGPLAN International conference on Functional programming. *SIGPLAN Not.* 48, 9, 391–402. https://doi.org/10.1145/2544174.2500592

Adam Chlipala. 2015. From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification. *SIGPLAN Not.* 50, 1 (Jan. 2015), 609–622. https://doi.org/10.1145/2775051.2677003

Adam Chlipala. 2018a. Formal reasoning about programs. http://adam.chlipala.net/frap/frap_book.pdf Course notes.

Adam Chlipala. 2018b. Formal reasoning about programs, Coq material for Chapter 14. https://github.com/achlipala/frap/blob/master/SeparationLogic.v

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2009. Effective Interactive Proofs for Higher-Order Imperative Programs. In *ACM International Conference on Functional Programming (ICFP)* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/1596550.1596565

David Costanzo and Zhong Shao. 2012. A Case for Behavior-Preserving Actions in Separation Logic. In *Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 332–349. https://doi.org/10.1007/978-3-642-35182-2_24

Robert Dockins, Andrew W. Appel, and Aquinas Hobor. 2008. Multimodal Separation Logic for Reasoning About Operational Semantics. *Electronic Notes in Theoretical Computer Science* 218 (2008), 5 – 20. https://doi.org/10.1016/j.entcs.2008.10.002 Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV).

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A Fresh Look at Separation Algebras and Share Accounting. In *Programming Languages and Systems*, Zhenjiang Hu (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 161–177. https://doi.org/10.1007/978-3-642-10672-9_13

Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. 2007. An Open Framework for Foundational Proof-Carrying Code. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)* (Nice, France). ACM Press, New York, NY, USA, 67–78. https://doi.org/10.1145/1190315.1190325

Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. 2008. Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 170–182. https://doi.org/10.1145/1375581.1375603

Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. 2006. Modular Verification of Assembly Code with Stack-Based Control Abstractions. *SIGPLAN Not.* 41, 6 (June 2006), 401–414. https://doi.org/10.1145/1133255.1134028

R. W. Floyd. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics, Vol. 19)*. American Mathematical Society, 19–32.

Michael J. C. Gordon. 1989. *Mechanizing Programming Logics in Higher Order Logic*. Springer-Verlag, Berlin, Heidelberg, 387–439. https://doi.org/10.1007/978-1-4612-3658-0_10

Alexey Gotsman, Josh Berdine, and Byron Cook. 2011. Precision and the Conjunction Rule in Concurrent Separation Logic. *Electronic Notes in Theoretical Computer Science* 276 (2011), 171 – 190. https://doi.org/10.1016/j.entcs.2011.09.021 Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. 2019. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *Interactive Theorem Proving (ITP) (Leibniz International Proceedings in Informatics, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:20. https://doi.org/10.4230/LIPIcs.ITP.2019.18

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *European Symposium on Programming (ESOP)*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 584–610. https://doi.org/10.1007/978-3-662-54434-1_22

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. http://doi.acm.org/10.1145/363235.363259

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *Programming Languages and Systems*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 353–367. https://doi.org/10.1007/978-3-540-78739-6_27

Aquinas Hobor, Robert Dockins, and Andrew W. Appel. 2010. A Theory of Indirection via Approximation. *SIGPLAN Not.* 45, 1 (Jan. 2010), 171–184. https://doi.org/10.1145/1707801.1706322

Aquinas Hobor and Jules Villard. 2013. The Ramifications of Sharing in Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2429069.2429131

Zhé Hóu, Rajeev Goré, and Alwen Tiu. 2015. Automated Theorem Proving for Assertions in Separation Logic with All Connectives. In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 501–516. https://doi.org/10.1007/978-3-319-21401-6_34

Zhé Hóu, David Sanán, Alwen Tiu, and Yang Liu. 2017. Proof Tactics for Assertions in Separation Logic. In *Interactive Theorem Proving*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.). Springer International Publishing, Cham, 285–303. https://doi.org/10.1007/978-3-319-66107-0_19

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. *SIGPLAN Not.* 36, 3 (Jan. 2001), 14–26. https://doi.org/10.1145/373243.375719

Jonas B. Jensen, Nick Benton, and Andrew Kennedy. 2013. High-Level Separation Logic for Low-Level Code. *SIGPLAN Not.* 48, 1 (Jan. 2013), 301–314. https://doi.org/10.1145/2480359.2429105

Jonas Braband Jensen and Lars Birkedal. 2012. Fictional Separation Logic. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 377–396. https://doi.org/10.1007/978-3-642-28869-2_19

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. *SIGPLAN Not.* 51, 9 (Sept. 2016), 256–269. https://doi.org/10.1145/3022670.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Andrew Kennedy, Nick Benton, Jonas B. Jensen, and Pierre-Evariste Dagand. 2013. Coq: The World's Best Macro Assembler?. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming* (Madrid, Spain) *(PPDP '13)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2505879.2505897

Gerwin Klein, Rafal Kolanski, and Andrew Boyton. 2012. Mechanised Separation Algebra. In *Interactive Theorem Proving*, Lennart Beringer and Amy Felty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 332–337. https://doi.org/10.1007/978-3-642-32347-8_22

Rafal Kolanski and Gerwin Klein. 2009. Types, Maps and Separation Logic. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 276–292. https://doi.org/10.1007/978-3-642-03359-9_20

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.* 2, ICFP, Article 77 (July 2018), 30 pages. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. Springer-Verlag, Berlin, Heidelberg, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Neelakantan R. Krishnaswami. 2012. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. Ph.D. Dissertation. USA. Advisor(s) Aldrich, Jonathan. https://doi.org/10.5555/2519942

Neelakantan R. Krishnaswami, Jonathan Aldrich, and Lars Birkedal. 2007. Modular verification of the subject-observer pattern via higher-order separation logic. In *In Proceedings of Formal Techniques for Java-like Programs (FTfJP)*.

Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. 2010. Verifying Event-Driven Programs Using Ramified Frame Properties. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Madrid, Spain) *(TLDI '10)*. Association for Computing Machinery, New York, NY, USA, 63–76. https://doi.org/10.1145/1708016.1708025

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. https://doi.org/10.1145/2535838.2535841

Peter Lammich. 2019a. Generating Verified LLVM from Isabelle/HOL. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:19. https://doi.org/10.4230/LIPIcs.ITP.2019.22

Peter Lammich. 2019b. Refinement to Imperative HOL. *Journal of Automated Reasoning (JAR)* 62, 4 (April 2019), 481–503. https://doi.org/10.1007/s10817-017-9437-1

Wonyeol Lee and Sungwoo Park. 2014. A Proof System for Separation Logic with Magic Wand. *SIGPLAN Not.* 49, 1 (Jan. 2014), 477–490. https://doi.org/10.1145/2578855.2535871

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. 2006. Formal Verification of the Heap Manager of an Operating System Using Separation Logic. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering* (Macao, China) *(ICFEM'06)*. Springer-Verlag, Berlin, Heidelberg, 400–419. https://doi.org/10.1007/11901433_22

Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. 2007. A General Framework for Certifying Garbage Collectors and Their Mutators. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 468–479. https://doi.org/10.1145/1250734.1250788

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Magnus O Myreen. 2008. *Formal verification of machine-code programs*. Ph.D. Dissertation.

Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare Logic for Realistically Modelled Machine Code. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Braga, Portugal) *(TACAS'07)*. Springer-Verlag, Berlin, Heidelberg, 568–582. https://doi.org/10.1007/978-3-540-71209-1_44

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 11423)*, Luis Caires (Ed.). Springer, 1–27. https://doi.org/10.1007/978-3-030-17184-1_1

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and Separation in Hoare Type Theory. *SIGPLAN Not.* 41, 9 (Sept. 2006), 62–73. https://doi.org/10.1145/1160074.1159812

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2008a. Hoare Type Theory, Polymorphism and Separation. *J. Funct. Program.* 18, 5–6 (Sept. 2008), 865–911. https://doi.org/10.1017/S0956796808006953

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008b. Ynot: Dependent Types for Imperative Programs. *SIGPLAN Not.* 43, 9 (Sept. 2008), 229–240. https://doi.org/10.1145/1411203.1411237

Zhaozhong Ni and Zhong Shao. 2006. Certified Assembly Programming with Embedded Code Pointers. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '06)*. Association for Computing Machinery, New York, NY, USA, 320–333. https://doi.org/10.1145/1111037.1111066

Zhaozhong Ni, Dachuan Yu, and Zhong Shao. 2007. Using XCAP to Certify Realistic Systems Code: Machine Context Management. In *Theorem Proving in Higher Order Logics*, Klaus Schneider and Jens Brandt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 189–206. https://doi.org/10.1007/978-3-540-74591-4_15

O'Hearn, Reynolds, and Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL: 15th Workshop on Computer Science Logic*. LNCS, Springer-Verlag. https://doi.org/10.1007/3-540-44802-0_1

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968 The appendix is linked as supplementary material from the ACM digital library.

Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *The Bulletin of Symbolic Logic* 5, 2 (1999), 215–244. http://www.jstor.org/stable/421090

Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. 2014. Software Verification with VeriFast: Industrial Case Studies. *Sci. Comput. Program.* 82 (March 2014), 77–97. https://doi.org/10.1016/j.scico.2013.01.006

Benjamin C. Pierce and many contributors. 2016. Software Foundations. https://softwarefoundations.cis.upenn.edu/

François Pottier. 2008. Hiding local state in direct style: a higher-order anti-frame rule. In *IEEE Symposium on Logic In Computer Science (LICS)*. Pittsburgh, Pennsylvania, 331–340. https://doi.org/10.1109/LICS.2008.16

François Pottier. 2017. Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic. In *ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)* (Paris, France) *(CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/3018610.3018624

Viorel Preoteasa. 2006. Mechanical Verification of Recursive Procedures Manipulating Pointers Using Separation Logic. In *FM 2006: Formal Methods*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 508–523. https://doi.org/10.1007/11813040_34

Bernhard Reus and Jan Schwinghammer. 2006. Separation Logic for Higher-Order Store. In *Computer Science Logic*, Zoltán Ésik (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 575–590. https://doi.org/10.1007/11874683_38

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Annual IEEE Symposium on Logic in Computer Science (LICS)*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

John C Reynolds. 2006. A short course on separation logic. http://cs.ioc.ee/yik/schools/win2006/reynolds/estslides.pdf

Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. 2019. A Verified Generational Garbage Collector for CakeML. *Journal of Automated Reasoning (JAR)* 63 (2019). https://doi.org/10.1007/s10817-018-9487-z

Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. 2009. Nested Hoare Triples and Frame Rules for Higher-Order Store. In *Computer Science Logic*, Erich Grädel and Reinhard Kahle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 440–454. https://doi.org/10.1007/978-3-642-04027-6_32

Jan Schwinghammer, Hongseok Yang, Lars Birkedal, François Pottier, and Bernhard Reus. 2010. A Semantic Foundation for Hidden State. In *Foundations of Software Science and Computational Structures*, Luke Ong (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–17. https://doi.org/10.1007/978-3-642-12032-9_2

Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. 2010. Verifying Generics and Delegates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–199. https://doi.org/10.1007/978-3-642-14107-2_9

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, Bytes, and Separation Logic. *SIGPLAN Not.* 42, 1 (Jan. 2007), 97–108. https://doi.org/10.1145/1190215.1190234

Thomas Tuerk. 2010. Local Reasoning about While-Loops. In *In International Conference on Verified Software: Theories, Tools and Experiments*.

Thomas Tuerk. 2011. *A separation logic framework for HOL.* Technical Report UCAM-CL-TR-799. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-799.pdf

Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

Carsten Varming and Lars Birkedal. 2008. Higher-Order Separation Logic in Isabelle/HOLCF. *Electronic Notes in Theoretical Computer Science* 218 (2008), 371 – 389. https://doi.org/10.1016/j.entcs.2008.10.022 Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV).

Wei Wang, Zhong Shao, Xinyu Jiang, and Yu Guo. 2011. A Simple Model for Certifying Assembly Programs with First-Class Function Pointers. In *5th IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE 2011, Xi'an, China, 29-31 August 2011*, Zhenhua Duan and C.-H. Luke Ong (Eds.). IEEE Computer Society, 125–132. https://doi.org/10.1109/TASE.2011.16

Tjark Weber. 2004. Towards Mechanized Program Verification with Separation Logic. In *Computer Science Logic*, Jerzy Marcinkowski and Andrzej Tarlecki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 250–264. https://doi.org/10.1007/978-3-540-30124-0_21

Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A practical verification framework for preemptive OS kernels. In *International Conference on Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, Springer International Publishing, Cham, 59–79. https://doi.org/10.1007/978-3-319-41540-6_4

Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. 2003. Building Certified Libraries for PCC: Dynamic Storage Allocation. In *Programming Languages and Systems*, Pierpaolo Degano (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 363–379. https://doi.org/10.1007/3-540-36575-3_25

Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. 2004. Building Certified Libraries for PCC: Dynamic Storage Allocation. *Science of Computer Programming* 50, 1-3 (2004), 101–127. https://doi.org/10.1007/3-540-36575-3_25

A STATISTICS ON THE MINIMAL SOUNDNESS PROOF FOR SEPARATION LOGIC

	Number of definitions	Number of lemmas	Lines of Tactic defs.	Lines of Proofs
Syntax of the language	8			
Substitution and big-step semantics	2			
Tactic for heap equality and disjointness			5	
Extensionality axioms	2			
Core heap predicates	7			
Entailment	2	4	2	4
Properties of separating conjunction		9		21
Properties of other operators		11		14
Tactic for entailments		2	17	3
Lemmas for heap-manipulating primitives		4		13
Hoare triples and associated rules	1	14		41
Separation Logic and associated rules	1	15		21
Total	23	59	24	117

Fig. 3. Statistics for the Coq formalization

B PREDICATE EXTENSIONALITY

In proof assistants such as HOL or Isabelle/HOL, extensionality is built-in. In Coq, it needs to be either axiomatized, or derived from two more fundamental extensionality axioms: extensionality for functions and extensionality for propositions.

These standard axioms are formally stated as follows.

$$\begin{array}{lllllll}
\textsc{predicate-extensionality:} & \forall A. & \forall (P\,P' : A \to \mathsf{Prop}). & (P\,x \Leftrightarrow P'\,x) & \Rightarrow & (P = P') \\
\textsc{functional-extensionality:} & \forall A\,B. & \forall (f\,f' : A \to B). & (f\,x = f'\,x) & \Rightarrow & (f = f') \\
\textsc{propositional-extensionality:} & & \forall (P\,P' : \mathsf{Prop}). & (P \Leftrightarrow P') & \Rightarrow & (P = P')
\end{array}$$

In summary, one may take FUNCTIONAL-EXTENSIONALITY and PROPOSITIONAL-EXTENSIONALITY as axioms in Coq, then from these two derive PREDICATE-EXTENSIONALITY and, as a corollary, the antisymmetry rule for heap entailment (HIMPL-ANTISYM).

C EXAMPLE PROOFS OF REASONING RULES

C.1 Proof of A Structural Rule

To illustrate the kind of reasoning involved in the proof of structural rules, consider the proof of the combined consequence-frame rule.

LEMMA C.1 (COMBINED CONSEQUENCE-FRAME RULE).

$$\frac{H \vdash H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 \vdash Q}{\{H\}\ t\ \{Q\}}\ \textit{CONSEQUENCE-FRAME}$$

PROOF. The consequence rule is straightforward to establish for Hoare triples, based on the definition of entailment (Definitions 3.7 and 3.14) and of Hoare triples (Definition 5.1), in which the precondition appears as hypothesis and the postcondition as conclusion of a logical implication.

Let us prove the consequence-frame rule with respect to the interpretation of Separation Logic triples given in Definition 5.2.

The conclusion $\{H\}\ t\ \{Q\}$ is equivalent to $\forall H'.\ ^{\text{HOARE}}\{H \star H'\}\, t\, \{Q \star H'\}$. Consider a particular heap predicate H'. Invoking the premise $\{H_1\}\ t\ \{Q_1\}$ on the predicate $H_2 \star H'$ yields $^{\text{HOARE}}\{H \star (H_2 \star H')\}\, t\, \{Q \star\!\!\!\star (H_2 \star H')\}$. By the consequence rule of Hoare logic, to derive $^{\text{HOARE}}\{H \star H'\}\, t\, \{Q \star\!\!\!\star H'\}$, it suffices to establish the entailments $H \star H' \vdash H \star (H_2 \star H')$ and $Q \star\!\!\!\star (H_2 \star H') \vdash Q \star\!\!\!\star H'$.

To prove the first entailment, first exploit rule STAR-ASSOC to rewrite it as $H \star H' \vdash (H \star H_2) \star H'$, then observe that the resulting entailment follows from the premise $H \vdash H_1 \star H_2$ by rule STAR-MONOTONE-R. To prove the second entailment, let us begin by revealing the definition of the entailment for postconditions (recall Definition 3.14). The second entailment is equivalent to $(Q\, v) \star (H_2 \star H') \vdash (Q\, v) \star H'$. It follows from the premise $Q_1 \star\!\!\!\star H_2 \vdash Q$, which implies $(Q_1\, v) \star H_2 \vdash (Q\, v)$, by exploiting the rules STAR-ASSOC and STAR-MONOTONE-R just like for the first entailment. □

C.2 Proof of a Reasoning Rule for Terms

To illustrate the kind of reasoning involved in the proof of reasoning rules for terms, consider the case of sequences. The proof is two-step: first establish a Hoare logic reasoning rule for sequences, then derive its Separation Logic counterpart.

LEMMA C.2 (REASONING RULE FOR SEQUENCES IN HOARE LOGIC).

$$\frac{^{HOARE}\{H\}\, t_1\, \{\lambda v.\, H'\} \qquad ^{HOARE}\{H'\}\, t_2\, \{Q\}}{^{HOARE}\{H\}\, (t_1\,;\, t_2)\, \{Q\}}\ \textit{HOARE-SEQ}$$

PROOF. The evaluation rule for sequence is a simplified version of the evaluation rule for let-bindings, stated as shown below.

$$\frac{t_1/s \Downarrow v_1/s' \qquad t_2/s' \Downarrow v/s''}{(t_1\,;\, t_2)/s \Downarrow v/s''}\ \text{EVAL-SEQ}$$

Recall from Definition 5.1 the interpretation of Hoare triples. Consider a state s satisfying the precondition H, that is, such that $H\, s$ holds. The goal is to find v and s' such that $(t_1\,;\, t_2)/s \Downarrow v/s'$ and $Q\, v\, s'$ hold.

By the first premise $^{\text{HOARE}}\{H\}\, t_1\, \{\lambda v.\, H'\}$ applied that state s, there exists v_1 and s_1' such that $t_1/s \Downarrow v_1/s_1'$ and $(\lambda v.\, H')\, v_1\, s_1'$ hold. The latter simplifies to $H'\, s_1'$.

By the second premise $\{H'\}\ t_2\ \{Q\}$ applied to the state s_1', which satisfies the precondition H', there exists v and s' such that $t_2/s \Downarrow v/s'$ and $Q\, v\, s'$ hold. The latter corresponds to one half of the conclusion.

Applying the evaluation rule for sequence EVAL-SEQ to the judgments $t_1/s \Downarrow v_1/s_1'$ and $t_2/s \Downarrow v/s'$ yields $(t_1\,;\, t_2)/s \Downarrow v/s'$, which corresponds to the second half of the conclusion. □

LEMMA C.3 (REASONING RULE FOR SEQUENCES IN SEPARATION LOGIC).

$$\frac{\{H\}\ t_1\ \{\lambda v.\, H'\} \qquad \{H'\}\ t_2\ \{Q\}}{\{H\}\ (t_1\,;\, t_2)\ \{Q\}}\ \textit{SEQ}$$

PROOF. Recall from Definition 5.2 the interpretation of Separation Logic triples: $\{H\}\ (t_1\,;\, t_2)\ \{Q\}$ is equivalent to $\forall H''.\ ^{\text{HOARE}}\{H \star H''\}\, (t_1\,;\, t_2)\, \{Q \star\!\!\!\star H''\}$. Consider a particular heap predicate H''.

By the first premise $\{H\}\ t_1\ \{\lambda v.\, H'\}$ applied to that H'', one derives $^{\text{HOARE}}\{H \star H''\}\, t_1\, \{(\lambda v.\, H') \star\!\!\!\star H''\}$. In that judgment, the postcondition $(\lambda v.\, H') \star\!\!\!\star H''$ simplifies to $\lambda v.\, (H' \star H'')$.

By the second premise $\{H'\}\ t_2\ \{Q\}$ applied to the same H'', one derives $^{\text{HOARE}}\{H' \star H''\}\, t_2\, \{Q \star\!\!\!\star H''\}$.

Applying the Hoare logic reasoning rule for sequences (HOARE-SEQ from Lemma C.2) to the two judgments $^{\text{HOARE}}\{H \star H''\}\, t_1\, \{\lambda v.\, (H' \star H'')\}$ and $^{\text{HOARE}}\{H' \star H''\}\, t_2\, \{Q \star\!\!\!\star H''\}$ yields $^{\text{HOARE}}\{H \star H''\}\, (t_1\,;\, t_2)\, \{Q \star\!\!\!\star H''\}$, as required. □

D VERIFICATION OF THE INCREMENT FUNCTION

To illustrate the use of the reasoning rules, let us present the proof of the increment function. Consider the following implementation.

$$\begin{array}{lll} \mathsf{incr} & \equiv & \hat{\mu}\_.\lambda p.\ \mathsf{let}\ n = \mathsf{get}\ p\ \mathsf{in} \\ & & \quad \mathsf{let}\ m = (+)\ n\ 1\ \mathsf{in} \\ & & \quad \mathsf{set}\ p\ m \end{array}$$

Example D.1 (Verification of the increment function). The following statement holds.

$$\forall p\, n.\ \ \{p \hookrightarrow n\}\ (\mathtt{incr}\ p)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}$$

Proof. Consider particular values of p and n.

- Applying rule app leaves:

$$\{p \hookrightarrow n\}\ (\mathsf{let}\ n = \mathsf{get}\ p\ \mathsf{in}\ \mathsf{let}\ m = (+)\ n\ 1\ \mathsf{in}\ \mathsf{set}\ p\ m)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

- Applying rule let with Q' instantiated as $\lambda r.\ [r = n] \star (p \hookrightarrow n)$ leaves two subgoals. The first one is: $\{p \hookrightarrow n\}\ (\mathsf{get}\ p)\ \{\lambda r.\ [r = n] \star (p \hookrightarrow n)\}$. It is an instance of the rule get. The second subgoal is:

$$\forall v.\quad \{[v = n] \star p \hookrightarrow n\}\ (\mathsf{let}\ m = (+)\ v\ 1\ \mathsf{in}\ \mathsf{set}\ p\ m)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

- Introducing v, applying the rule prop to extract $[v = n]$ from the precondition, then substituting n for v turns the proof obligation into:

$$\{p \hookrightarrow n\}\ (\mathsf{let}\ m = (+)\ n\ 1\ \mathsf{in}\ \mathsf{set}\ p\ m)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

- Applying again the rule let, this time with Q' instantiated as $\lambda r.\ [r = n+1] \star (p \hookrightarrow n)$. leaves two subgoals. The first one is: $\{p \hookrightarrow n\}\ ((+)\ n\ 1)\ \{\lambda r.\ [r = n+1] \star (p \hookrightarrow n)\}$. Applying the frame rule with H' instantiated as $p \hookrightarrow n$ yields an instance of the rule add. The second subgoal is:

$$\forall v.\quad \{[v = n+1] \star p \hookrightarrow n\}\ (\mathsf{set}\ p\ v)\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

- Introducing and eliminating v simplifies the proof obligation into an instance of rule set:

$$\{p \hookrightarrow n\}\ (\mathsf{set}\ p\ (n+1))\ \{\lambda\_.\ p \hookrightarrow (n+1)\}.$$

□

E BENEFITS OF THE FRAME RULE IN THE PROOF OF A RECURSIVE FUNCTION

The frame rule allows for simpler proofs than what could be achieved without it. To substantiate this claim, let us present the proof of the copy function in Separation Logic, then discuss how it would be more complicated without the frame rule.

```
let rec mcopy p =
  if p == null
    then null
    else { head = p.head; tail = mcopy p.tail }
```

Proof. The proof of the specification stated in Example 2.12 is carried out by induction on the length of the list L. The induction hypothesis allows in particular to assume the specification to hold for the recursive call.

At the entry of the body of the function, the state corresponds to the precondition, that is, to $\mathsf{Mlist}\ L\ p$. To reason by case analysis on whether p is null, we exploit Definition 2.9.

- Case $p = \mathsf{null}$. In this case, $L = \mathsf{nil}$ and the function returns the value null. This value is named p' in the postcondition. The new piece of postcondition to establish is $\mathsf{Mlist}\, L\, p'$. By Definition 2.8, because $L = \mathsf{nil}$, the predicate $\mathsf{Mlist}\, L\, p'$ is equivalent to $p' = \mathsf{null}$. Hence, the postcondition is satisfied.
- Case $p \neq \mathsf{null}$. In this case, L decomposes as $x :: L'$, and the current state is described by the heap predicate $(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q)$. The operations performed by the code are verified as follows.
  - The read operation `p.head` returns the value x, and `p.tail` returns the address q.
  - The recursive call `mcopy` q creates a copy of the list represented by L'. By induction hypothesis applied to L', this recursive call can be assumed to satisfy the triple:

$$\{\mathsf{Mlist}\, L'\, q\}\ (\texttt{mcopy}\ q)\ \{\lambda r'.\ \exists q'.\ [r' = q'] \star (\mathsf{Mlist}\, L'\, q) \star (\mathsf{Mlist}\, L'\, q')\}.$$

  - Applying the frame rule to that triple and to $(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)$ yields the triple:

$$\begin{array}{l}\{(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q)\}\\ (\texttt{mcopy}\ q)\\ \{\lambda r'.\ \exists q'.\ [r' = q'] \star (p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q) \star (\mathsf{Mlist}\, L'\, q')\}\end{array}$$

    which enables reasoning about the recursive call in the current state.
  - Let q' denote the result of the recursive call, and let p' denote the result of the record allocation operation `{ head = p.head; tail = q' }`. This record allocation produces a heap described by $(q.\mathsf{head} \hookrightarrow x) \star (q.\mathsf{tail} \hookrightarrow q')$.
  - Thus, the final state is described by:

$$\begin{array}{c}(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q) \star (\mathsf{Mlist}\, L'\, q)\\ \star (p'.\mathsf{head} \hookrightarrow x) \star (p'.\mathsf{tail} \hookrightarrow q') \star (\mathsf{Mlist}\, L'\, q')\end{array}$$

    which may be folded to $(\mathsf{Mlist}\, L\, p) \star (\mathsf{Mlist}\, L\, p')$, matching the claimed postcondition. □

Intuitively, in the above reasoning, the frame rule is invoked at each recursive call on the head cell of the input list. One is therefore able to reason about a recursive call to `mcopy` by assuming that it makes a copy of a sublist, *independently* of all the cells that have already been traversed by the outer recursive calls to `mcopy`.

Without the frame rule, one would have to describe the full list at an arbitrary point during the recursion. Doing so requires describing the *list segment* made of cells ranging from the head of the initial list up to the pointer on which the current recursive call is made. Stating an invariant involving list segments is doable yet involves more complex definitions and assertions.

More generally, for a program manipulating tree-shaped data structures, the frame rule saves the need to describe a tree with a subtree carved out of it. The ability to invoke the frame rule in proofs carried out by induction allows to reason *locally* about the work performed by the recursive call, without having to explicitly describe the whole *context* in which this recursive call is taking place, thereby saving a significant amount of proof effort.

F REASONING ABOUT HIGHER-ORDER FUNCTIONS

Example F.1 (Reynold's CPS-append challenge). Consider the (nontrivial) function shown below. It takes as arguments two mutable lists, and returns an address describing the head of a list whose cells correspond to the concatenation of the two input lists. The function is implemented by means of a recursive function that expects a continuation, named `k` in the code, to be invoked on the output list. Each recursive call is made with a fresh continuation, responsible for updating the tail pointer of the list cell at hand before invoking the current continuation.

```
let cps_append p1 p2 =
  let rec aux p k =
    if p == null
      then k p2
      else aux p.tail p2 (fun r => (p.tail <- r); k p)
    in
 aux p1 (fun r => r)
```

This function admits the following Separation Logic specification, which describes the two disjoint input lists represented by L_1 and L_2, respectively, and the result list represented by $(L_1 +\!\!+ L_2)$.

$$\{(\mathsf{Mlist}\, L_1\, p_1) \star (\mathsf{Mlist}\, L_2\, p_2)\}\ (\mathsf{cps\_append}\, p_1\, p_2)\ \{\lambda r.\ \exists p_3\ [r = p_3] \star \mathsf{Mlist}\, (L_1 +\!\!+ L_2)\, p_3\}$$

The crux of the proof consists of providing the appropriate specification for the internal recusive function aux. This specification reads as follows. Assume the continuation k to admit the postcondition Q when invoked on a list p_3 describing a list of the form $L +\!\!+ L_2$, and with an auxiliary heap described by H. Then, the call $\mathsf{aux}\, p_1\, k$ also admits the postcondition Q (indeed, the ultimate action performed by aux is to invoke its continuation k) under the precondition including $(\mathsf{Mlist}\, L\, p)$, which corresponds to a sublist of $(\mathsf{Mlist}\, L_1\, p_1)$, including $(\mathsf{Mlist}\, L_2\, p_2)$, which is exploited only in the base case where p becomes null, and including H.

$$\begin{array}{lll} \forall p\, k\, L. & & \big(\forall p_3.\ \{\mathsf{Mlist}\, (L +\!\!+ L_2)\, p_3 \star H\}\ (k\, p_3)\ \{Q\}\big) \\ & \Rightarrow & \{(\mathsf{Mlist}\, L\, p) \star (\mathsf{Mlist}\, L_2\, p_2) \star H\}\ (\mathsf{aux}\, p_1\, k)\ \{Q\} \end{array}$$

In this statement, H describes, in an abstract way, the cells from list p_1 that have already passed by. The introduction of such an abstract heap predicate H is a common pattern for CPS-style functions.

The verification of the function cps_append was first conducted in XCAP [Ni and Shao 2006]. In this pioneering work, the function is programmed in a assembly-level language with embedded code pointers, and the proof involving several hundreds of lines of Coq script. The verification of cps_append was later carried out in CFML [Charguéraud 2011]. In that framework, the function is programmed in 4 lines of OCaml and verified using 9 lines of Coq proofs.

G ALTERNATIVE STRUCTURAL RULES

Lemma 5.4 presents 4 core structural reasoning rules: CONSEQUENCE, FRAME, PROP and EXISTS.

The rule PROP for extracting pure facts may in fact be seen as a particular instance of the rule EXISTS for extracting existential quantifiers. Indeed, as pointed out in Remark 3.4, the heap predicate $[P]$ is equal to $\exists(p : P).\ [\,]$. Besides, the quantification "$\forall(p : P).\ \ldots$" is equivalent to the implication "$P \Rightarrow \ldots$".

The rule EXISTS is very useful in practice, although its statement does not appear in the original papers on Separation Logic. These papers instead formulated a rule featuring an existential quantifier both in the precondition and the postcondition. In a ML-style language, it would corresponds to the rule EXISTS2 shown below. The rules EXISTS and EXISTS2 yield equivalent expressive power, that is, they may be derived from one another (in the presence of the rule CONSEQUENCE, and EXISTS-R and EXISTS-L from Fig. 1). Compared with EXISTS2, the statement of EXISTS is more concise and better-suited for practical purpose.

The rule FORALL, stated below, is also useful in practice. This rule is derivable from the CONSEQUENCE, and FORALL-L from Fig. 1.

Lemma G.1 (Other structural rules).

$$\frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\ H\}\ t\ \{\lambda v.\ \exists x.\ (Q\ v)\}}\ \textit{EXISTS2} \qquad \frac{\{[a/x]\ H\}\ t\ \{Q\}}{\{\forall x.\ H\}\ t\ \{Q\}}\ \textit{FORALL}$$

H ARRAYS AND RECORDS

This section briefly summarizes the key ideas involved in the treatment of arrays and records. Details may be found in the accompanying Coq development.

Note that the presentation does not take into account the notion of "allocated blocks", and the usual restriction that the free operation can only be invoked on the head of an allocated block. We are currently working on a refinement of the definitions that would enforce this property.

H.1 Arrays

The programming language is assumed to include two additional primitive operations: alloc n for allocating n consecutive cells, and dealloc $n\,p$ for deallocating n consecutive cell starting from address p. It is possible to refine the Separation Logic to ensure that deallocation operations are only invoked on the head of allocated blocks—details are beyond the scope of the present paper.

The allocated cells are assigned as contents a special *uninitialized value*, written $\bot$. The semantics of read operations may be adapted to prevent read operations in uninitialized cells, by adding a premise of the form $v \neq \bot$ to the specification of get.

The i-th cell of an array allocated at address p corresponds to the cell at address $p + i$.

Definition H.1 (Representation of consecutive cells). The heap predicate cells $L\,p$ describes consecutive cells allocated in the range $[p, p + |L|)$, whose contents are the items from the list L.

$$\begin{array}{rcl} \mathsf{cells}\,L\,p & \equiv & \mathsf{match}\,L\,\mathsf{with}\ |\ \mathsf{nil} \Rightarrow [\,] \\ & & \qquad\qquad |\ x :: L' \Rightarrow (p \hookrightarrow x)\ \star\ (\mathsf{cells}\,L'\,(p+1)) \end{array}$$

Lemma H.2 (Specification of array operations).

$$\begin{array}{rcllll} n \geq 0 & \Rightarrow & \{[\,]\} & (\mathit{alloc}\,n) & \{\lambda r.\ \exists p.\ [r = p]\ \star\ \mathit{cells}\,(\mathit{List.make}\,n\,\bot)\,p\} \\ n = |L| & \Rightarrow & \{\mathit{cells}\,L\,p\} & (\mathit{dealloc}\,n\,p) & \{\lambda\_.\ [\,]\} \\ 0 \leq i < |L| & \Rightarrow & \{\mathit{cells}\,L\,p\} & (\mathit{array\_get}\,i\,p) & \{\lambda r.\ [r = \mathit{List.nth}\,i\,L]\ \star\ \mathit{cells}\,L\,p\} \\ 0 \leq i < |L| & \Rightarrow & \{\mathit{cells}\,L\,p\} & (\mathit{array\_set}\,i\,v\,p) & \{\lambda\_.\ \mathit{cells}\,(\mathit{List.update}\,i\,v\,L)\,p\} \end{array}$$

For functions that process an array by making recursive calls to increasingly-smaller segments of the array, the following *range-split* lemma allows splitting the segment at hand and applying the frame rule to the segment that is not involved in the recursive call. One thereby gets for free the fact that cells in that segment are unmodified during the recursive call.

Lemma H.3 (Splitting a range of cells).

$$(\mathit{cells}\,(L_1 +\!\!+ L_2)\,p) \ =\ (\mathit{cells}\,L_1\,p)\ \star\ (\mathit{cells}\,L_2\,(p + |L_1|))$$

Another useful result is the following *focus* lemma, which allows isolating the i-th cell out of a range of consecutive cells starting at address p and described by a list L, so as to perform operations on that cell in isolation from the rest of the range. Subsequently, the cell with its updated contents v may be merged back into the range. This logical operation involves cancelling a magic wand.

Lemma H.4 (Focusing on a cell from a range). *Assume* $0 \leq i < |L|$.

$$\Big(\mathit{cells}\,L\,p\Big)\ \vdash\ \Big((p+i) \hookrightarrow (\mathit{List.nth}\,i\,L)\Big)\ \star\ \Big(\forall v.\ ((p+i) \hookrightarrow v)\ \mathrel{-\!\!\star}\ \mathit{cells}\,(\mathit{List.update}\,i\,v\,L)\,p\Big)$$

H.2 Records

Definition H.5 (Representation of record fields). A record field is represented by the heap predicate $p.k \hookrightarrow v$, which stands for $(p + k) \hookrightarrow v$.

Definition H.6 (Representation of records). Consider for example the record `{head = x; tail = q}`, with the offsets head ≡ 0 and tail ≡ 1. This record may be represented in three different ways:

(1) as "cells $(x :: q :: \mathsf{nil})\, p$", just like an array of length 2,
(2) as "$(p.\mathsf{head} \hookrightarrow x) \star (p.\mathsf{tail} \hookrightarrow q)$", with two separated fields,
(3) as "record $(\mathsf{head}, x) :: (\mathsf{tail}, q) :: \mathsf{nil})\, p$", where the heap predicate record $K\, p$ describes a record at location p and whose field names and contents are described by the association list K.

$$\begin{array}{lcl} \mathsf{record}\, K\, p & \equiv & \mathsf{match}\, K\, \mathsf{with} \mid \mathsf{nil} \Rightarrow [\,] \\ & & \qquad\qquad \mid (k, v) :: K' \Rightarrow (p.k \hookrightarrow v) \star (\mathsf{cells}\, K'\, p) \end{array}$$

To each of these three representations of records correspond different specifications for allocation, deallocation, read and write operations. The third representation is the one that scales better to large programs, because: (1) it groups the fields of a same record into a single predicate, thereby reducing the number of conjuncts, (2) it allows describing the ownership of an arbitrary subset of the fields, (3) it allows providing items in any order, and (4) with appropriate syntactic sugar it may be written in the form $p \hookrightarrow \{\mathsf{head} := x; \mathsf{tail} := q\}$, which is easy to read.

I TREATMENT OF ASSERTIONS (DYNAMIC CHECKS)

The language construct "assert t" expresses a Boolean assertion. If the term t evaluates to the value true, the assertion produces unit. Otherwise, the term "assert t" gets stuck—the program halts on an error. The verification of a program should statically ensure that: (1) the body of every assertion evaluates to true, and (2) the program remains correct when assertions are disabled either via a compiler option such as `-noassert` in OCaml, or via the programming pattern "if debug then assert t", where debug denotes a compilation flag. The ASSERT rule, shown below, satisfies these two properties.

LEMMA I.1 (EVALUATION RULES AND REASONING RULE FOR ASSERTIONS).

$$\frac{t/s \Downarrow \mathit{true}/s'}{(\mathit{assert}\, t)/s \Downarrow tt/s'}\ \textit{EVAL-ASSERT-ENABLED} \qquad \frac{}{(\mathit{assert}\, t)/s \Downarrow tt/s}\ \textit{EVAL-ASSERT-DISABLED} \qquad \frac{\{H\}\ t\ \{\lambda r.\ [r = \mathit{true}] \star H\}}{\{H\}\ (\mathit{assert}\, t)\ \{\lambda\_.\ H\}}\ \textit{ASSERT}$$

Remark I.2 (Assert false). The term "assert false" denotes inaccessible branches of the code. A valid triple for this term can only be derived from a false precondition: {[False]} (assert false) {Q}.

Remark I.3 (Assertions involving write operations). Interestingly, the reasoning rule ASSERT is not limited to read-only terms. For example, consider the Union-Find data structure, which involves the operation `find` that performs path compression. The evaluation of an assertion of the form `assert (find x = find y)` may involve write operations. It nevertheless preserves all the invariants of the data structure. These invariants would be captured by the heap predicate H from rule ASSERT.

J TREATMENT OF FUNCTIONS OF SEVERAL ARGUMENTS

Functions of several arguments may be represented as curried functions (`fun x y => t`), as tupled functions (`fun (x,y) => t`), or as native n-ary functions (like, e.g., in the C language),

The curried function $\mu f.\lambda x_1.\lambda x_2.\, t$ is represented as $\mu f.\lambda x_1.(\mu\_.\lambda x_2.t)$. An application takes the form $(v_0\, v_1\, v_2)$. The reasoning rule for such an application, APP2, generalizes the rule APP. A version of this rule may be stated for every arity. Alternatively, an arity-generic rule may be devised, although in practice it requires tactic support for synthesizing the lists of variables and arguments.

Lemma J.1 (Reasoning rule for curried functions of arity 2).

$$\frac{v_0 = \hat{\mu} f.\lambda x_1 x_2.t \qquad \{H\}\ ([v_2/x_2]\,[v_1/x_1]\,[v_0/f]\,t)\ \{Q\} \qquad \mathit{noduplicates}\,(f :: x_1 :: x_2 :: \mathit{nil})}{\{H\}\ (v_0\, v_1\, v_2)\ \{Q\}}\ \mathit{APP2}$$

A language featuring primitive n-ary functions more naturally admits arity-generic reasoning rules. In such a language, $\mu f.\lambda \overline{x}.t$ denotes a function f expecting a list $\overline{x}$ of arguments of the form "$x_1 :: \ldots :: x_n :: \mathsf{nil}$", and the n-ary application term $v_0\,\overline{v}$ denotes the application of a value v_0 to a list of arguments $\overline{v}$ of the form "$v_1 :: \ldots :: v_n :: \mathsf{nil}$".

Lemma J.2 (Reasoning rule for primitive n-ary functions).

$$\frac{v_0 = \hat{\mu} f.\lambda \overline{x}.t \qquad \{H\}\ [(v_0 :: \overline{v})/(f :: \overline{x})]\,t\ \{Q\} \qquad |\overline{v}| = |\overline{x}| > 0 \qquad \mathit{noduplicates}\,(f :: \overline{x})}{\{H\}\ (v_0\, \overline{v})\ \{Q\}}\ \mathit{APPS}$$

Remark: it is possible to set up coercions in Coq such that an application written in curried style $v_0\, v_1 \ldots v_n$ gets interpreted as the n-ary application $v_0\,(v_1 :: \ldots :: v_n :: \mathsf{nil})$.

K A TACTIC FOR SIMPLIFYING ENTAILMENTS

In practical proofs, entailment relations to be established typically involve dozens of tokens, e.g.:

$$\exists v.\,(q \hookrightarrow v) \star [n = 4] \star (p \hookrightarrow n) \star H \ \vdash\ \exists m.\,(p \hookrightarrow m) \star H \star [m > 0] \star \top$$
$$H1 \star H2 \star \Big((H1 \star H3) \mathrel{-\!\!\star} (H4 \mathrel{-\!\!\star} H5)\Big) \star H4 \ \vdash\ \Big((H2 \mathrel{-\!\!\star} H3) \mathrel{-\!\!\star} H5\Big).$$

Proving such entailment relations by manually invoking the appropriate reasoning rules involves an overwhelming amount of work. Thus, for practical program verification, automation is a must-have. Ideally, an automated procedure should not only be able to discharge true entailments, it should also be able to perform all the "obvious" simplifications, leaving what remains of the entailment for a manual processing step by the user. Nearly all practical verification framework come with some amount of tooling for simplifying entailments.

One may also wish that the simplification tactic systematically produces as output an entailment that is logically equivalent to its input. However, there are a number of simplifications that are technically not equivalence-preserving, yet nearly always desirable to perform. It seems to make sense to nevertheless apply these simplifications, taking into account that the user always has the ability to "lock" certain subexpressions for preventing simplifications involving them.

In what follows, we describe (in informal terms) a strategy for simplifying triples in a systematic and predictable manner. The corresponding tactic is named `xsimpl` in the Coq development. (It is entirely implemented in Ltac.) The tactic named `xpull` is a variant of `xsimpl` that is limited to performing simplifications only on the left-hand side.

Definition K.1 (Tactic for simplifying entailment). The tactic first attempts to perform simplifications in the left-hand side, then in the right-hand side, then it attempts to perform simplifications that involve both sides. It may iterate this process several times, as long as progress is made.

(1) On each of the two sides, independently:
   - Remove empty heap predicates and normalize expressions with respect to associativity.
   - Collapse occurrences of the predicate $\top$, so that there is at most one occurrence at top-level on each side.
   - Bring all existential quantifiers and pure facts to the front of each side.

(2) On the left-hand side:
   - Pull out pure facts and existential quantifiers out into the Coq context.
   - For each magic wand of the form $(H_1 \star \ldots H_n) \mathrel{-\!\!\star} H'$, if one of the H_i also occurs on the left-hand side, cancel H_i out as explained Lemma 7.3.

- For each magic wand of the form $Q \mathrel{-\!\!\star} Q'$, if a predicate of the form $Q\,v$ also occurs on the left-hand side, then specialize this magic wand to $Q\,v \mathrel{-\!\!\star} Q'\,v$, cancel out $Q\,v$, leaving $Q'\,v$.

(3) On the right-hand side:
- If a pure fact $[P]$ occurs on the right-hand side, remove it and generate a subgoal asserting the proposition P.
- It a quantifier $\exists x.\, H$ occurs on the right-hand side, instantiate x with a value v, leaving $[v/x]\,H$. The value v may be provided by the user (via arguments passed to the tactic `xsimpl`), or it may be realized as a Coq unification variable (a.k.a. *evar*).
- Remove expressions of the form $H \mathrel{-\!\!\star} H$ or $Q \mathrel{-\!\!\star} Q$, which are entailed by $[\,]$.

(4) On both sides:
- If a predicate H occurs on the both sides, and H is not $\top$, then remove H from both sides.
- If the entailment is of the form $H_0 \vdash (H_1 \mathrel{-\!\!\star} H_2)$, then replace it with $(H_1 \star H_0) \vdash H_2$. Likewise, replace $H_0 \vdash (Q_1 \mathrel{-\!\!\star} Q_2)$ with $(Q_1 \star H_0) \vdash Q_2$.
- If the entailment is of the form $[\,] \vdash [\,]$, then the entailment is true.
- It the entailment is of the form $H \vdash \top$, replace it with the proof obligation $\mathsf{affine}\,H$.
- If the entailment is of the form $H \vdash (\forall x.\, H')$, replace it with the proof obligation $\forall x.\, (H \vdash H')$ and continue simplifying the inner entailment.
- If the goal is of the form $Q \vdash Q'$, replace it with the proof obligation $\forall x.\, (Q\,x \vdash Q'\,x)$ and continue simplifying the inner entailment.

Several papers investigate specifically the automation of entailment simplification or resolution, e.g., [Hóu et al. 2015, 2017; Lee and Park 2014].